UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

AD-A196 188

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88-119	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FINITE PRECISION ARITHMETIC IN SINGULAR VALUE DECOMPOSITION ARCHITECTURES		5. TYPE OF REPORT & PERIOD COVERED PHD ~~MS~~ THESIS
		6. PERFORMING ORG. REPORT NUMBER
AUTHOR(s) ROBERT ARTHUR DURYEA		8. CONTRACT OR GRANT NUMBER(s)
PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: CORNELL UNIVERSITY		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988
		13. NUMBER OF PAGES 220
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

DTIC ELECTE AUG 0 3 1988 S D

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

D

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER [illegible] 20 July 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FINITE PRECISION ARITHMETIC IN SINGULAR VALUE DECOMPOSITION ARCHITECTURES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Robert Arthur Duryea

August 1987

-A-

FINITE PRECISION ARITHMETIC IN SINGULAR VALUE DECOMPOSITION ARCHITECTURES

Robert Arthur Duryea, Ph.D.

Cornell University 1987

The singular value decomposition (SVD) is an important matrix algorithm which has many applications in signal processing. However, its use has been limited due to its computational complexity. Complex ~~Several~~ architectures have been proposed to compute the SVD using arrays of parallel processors. ~~In~~ this thesis ~~we~~ derive requirements for the precision of arithmetic units (AUs) used in SVD arrays and compare the resource requirements of several architectures.

The ~~Our~~ results are based on the assumption that we are operating on matrices of quantized data. Since the matrices have quantization errors, it is shown we show that their singular values will have quantization errors which are as large as the data errors. To compute the number of bits needed in SVD AUs, it is required ~~we require~~ that the AUs have enough bits to keep the round-off errors of the SVD computation smaller than the quantization errors.

The ~~Our~~ analysis shows that we need essentially the same number of bits for either the Hestenes or Jacobi SVD algorithms. If the matrix has been scaled to prevent overflows and if we use properly rounded arithmetic, CORDIC and fixed point AUs require 8 fewer bits than floating point AUs. Our computations indicate that 32 bit floating point AUs are useful only for small arrays of 8-bit data. For 100-by-100 arrays of 16-bit data we need 40-bit floating point AUs. 32-bit fixed point AUs can be used in SVD arrays for large 8-bit matrices or moderate size 16-bit arrays. [illegible]

[illegible]

We describe five SVD architectures, two "linear" structures and three "quadratic" arrays, and compare their resource requirements with floating point and CORDIC AUs. Our comparison shows the total resource requirements of the linear designs to be lower than that of the quadratic arrays for all size matrices. The speed of the linear structures is competitive with the quadratic arrays for matrices up to size 200-by-200 even though the linear designs require many fewer AUs. CORDIC AUs simplify the architectures but they double the resource requirements and increase the computation times. We conclude that a linear array with floating point or fixed point AUs is the best design for implementation with current VLSI technology.

BIOGRAPHICAL SKETCH

Robert Arthur Duryea was born on February 13, 1951 in the village of South Egremont, Massachusetts. He graduated in 1968 from Mount Everett Regional High School in Sheffield, Massachusetts. He entered Rensselaer Polytechnic Institute (RPI) in the Fall of 1968. Four years later he graduated Cum Laude with a Bachelor's Degree in Electrical Engineering. He simultaneously earned a commission as a 2Lt from the Reserve Officer Training Corps of the United States Air Force. He remained at RPI for one more year and obtained a Master of Science degree in Management in 1973. He then went on active duty in the Air Force and has remained on active duty, progressing to the rank of Major.

His Air Force career has consisted of a series of jobs in research and development with periodic returns to college. His first assignment was to the Rome Air Development Center at Griffiss AFB, New York where he developed systems for compressing and transmitting imagery. In 1979 he obtained a Master of Science Degree in Electrical Engineering at the Air Force Institute of Technology at Wright-Patterson AFB, Ohio. Then he was assigned to the Air Force Technical Application Center at Patrick AFB, FLorida where he developed computer systems to process seismic and hydroacoustic data. Following this assignment, the Air Force offered him a once in a life time opportunity to obtain a Ph.D. in Electrical Engineering at their expense at the civilian school of his choice. After the completion of his Cornell Ph.D. program he will be assigned to the Air Force Weapons Laboratory at Kirtland AFB, New Mexico.

Robert is married to the former Marie Malnati. They are the proud parents of two sons: Andrew, age 10, and Ryan, age 5.

DEDICATION

To my wife, Marie, and sons, Andrew and Ryan

Thank you for your patience and understanding

I love you all very much

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to Professor Christopher Pottle, the chairman of my special committee. His constant encouragement, guidance and support where crucial to the completion of this thesis. I am especially grateful for the understanding he has shown for the special circumstances of a married graduate student with children and a compressed time schedule.

My thanks to Professor Franklin Luk whose courses and research provided the basis for the majority of my research. I would also like to thank Professor Gianfranco Bilardi whose courses on VLSI architectures and complexity theory sparked my interest in this area of research. I was extremely fortunate to have both of these individuals on my research committee since they are at the very forefront of VLSI/digital signal processing research.

I would like to thank the members of Professor Luk's research group for the comments and suggestions they have made on my work. I would especially like to thank Magnus Ewerbring for the unqualified friendship and support he has given me through all of my course work and research.

Finally I would like to thank the Air Force and the taxpayers of the United States for sponsoring this program.

Table of Contents

Chapter/Section Page

Table of Contents (continued)

Table of Contents (continued)

List of Tables

List of Figures

List of Figures (continued)

1.0 INTRODUCTION

The singular value decomposition (SVD) is a very important matrix algorithm which has many potential applications in signal processing. However its use has been limited, particularly in real-time processing, due to its computational complexity. As a result there has been a great amount of research interest in fast, parallel implementations of the SVD. To date little of this research has been translated into hardware. The purpose of this thesis is to derive requirements for the precision of arithmetic units used in SVD processors and to compare the resource requirements of several of the proposed architectures. The comparison is done from an engineering design point of view. That is, the architectures are analyzed in detail with an eye toward actual implementation in very large scale integrated (VLSI) circuits.

1.1 Definition and Applications of the SVD

The singular value decomposition (SVD) of a real m-by-n matrix A is given by

$$A = U \Sigma V^T \tag{1.1.1}$$

where U is an orthogonal m-by-m matrix; V is an orthogonal n-by-n matrix; and Σ is a nonnegative "diagonal" m-by-n matrix.

The SVD is a powerful matrix algorithm. Much of the interest in the SVD has been at a theoretical level since it provides a fundamental representation of the properties of a matrix. For example, the singular values can be used to determine the norms, rank and condition number of a matrix. The singular vectors provide bases for the range and null space of the original matrix [see Gol83 for details]. The SVD is also intimately related to the symmetric eigenproblem since the singular values are the square roots of the eigenvalues of A^TA or AA^T and the singular vectors are the eigenvectors of these symmetric matrices.

In this thesis we will be concerned not with the theoretical characteristics of the SVD but with its practical use. In particular we are interested in signal processing applications of the SVD. Normally, in these types of applications the SVD is used to decompose large matrices of sensor data to extract fundamental information about signals contained in the data. We often want to be able to perform the SVD repeatedly and rapidly as data flows in from the sensors. These types of applications have been the driving force behind the development of the sophisticated parallel architectures for computing the SVD which we will analyze in later chapters. Applications of the SVD include:

1. Image processing and compression [see for example Shi81, And76]
2. Adaptive beamforming and signal enhancement [Spe83]
3. Solution of least squares problems and systems of equations [Gol83]

1.2 Need for Parallel Architectures to Compute the SVD

While there are many potential applications for the SVD, few have been brought to fruition due to the computational complexity of the algorithms used to compute the decomposition. If we are decomposing an n-by-n array of data, even the most efficient SVD algorithm, the Golub-Reinsch algorithm [Gol70], requires order n^3 [$O(n^3)$] operations. Many of the practical applications of the SVD have been experimentally demonstrated only on mainframe computers using standard math library implementations of the Golub-Reinsch algorithm. Even at that, investigators are limited to operating on small matrices (normally smaller than 50-by-50) due to the long computation times involved in computing the SVD of larger arrays.

The majority of signal processing applications do not allow data to be collected at a central location and processed in batch mode on mainframe computers. Typical applications require the data to be processed by equipment which is small, light, rugged and power efficient. The data will usually be flowing in continuously and must be processed in real time. Additionally, many SVD applications require the handling of large arrays (as many as 6000-by-6000 data values for image processing). These requirements can only be met by special purpose parallel architectures which are implemented in VLSI circuits.

1.3 Need for Comparison of SVD Architectures

There have been a number of different parallel architectures proposed to compute the SVD. Some involve linear arrays of O(n) processors which compute the SVD in nearly $O(n^2)$ time. Others use $O(n^2)$ arrays of processors and can compute the SVD in nearly O(n) time. However, these architectures have been described only at a very high theoretical level. There has been little engineering analysis of the proposed designs to determine what is required to translate them into hardware. In particular no one has compared the different architectures to determine which would be the best for implementation with current VLSI technology. One of the primary objectives of this thesis is to analyze and compare several of the proposed SVD architectures.

1.4 Use of Finite Precision Arithmetic

The second major objective of this thesis is to derive requirements for the arithmetic units used to construct SVD arrays. Specifically we want to determine how many bits are needed in the data words used to store, transmit and process values in SVD hardware. Again we will have signal processing applications in mind.

Usually, in the signal processing world, the matrices we deal with are filled with quantized data values produced by "digital" sensors or A-to-D converters. As a result the data values have inherent "quantization" errors as will the singular values and vectors produced by an SVD array. Accordingly it is not necessary to compute the SVD of such matrices to high precision. There has been much theoretical work on the round-off error characteristics of SVD algorithms. We will utilize the results of this work to determine the number of bits needed in finite precision arithmetic units to guarantee that the SVD output is as accurate as the quantization error will allow. We will also investigate the use of different types of arithmetic including standard floating point and fixed point math and the rotation based CORDIC arithmetic proposed by Volder [Vol59].

1.5 Organization of the Thesis

This thesis is organized as follows. Chapter 2 lays the groundwork for the subsequent analyses by describing the two known methods for computing the SVD in parallel, the Jacobi and Hestenes algorithms. It also describes the characteristics of the input matrices which are commonly encountered in signal processing applications of the SVD. Chapter 3 gives the requirements which finite precision AUs must meet to compute the SVD of quantized data matrices.

In Chapter 4 we present a theoretical analysis of the errors encountered in computing the SVD with the Jacobi algorithm. Chapter 5 gives the results of computer simulations of the Jacobi algorithm computed with finite precision arithmetic. The theoretical analysis and computer simulation of the Hestenes algorithm are presented in Chapter 6. Finally Chapter 7 combines the results of Chapters 3, 4, 5 and 6 to develop expressions for the number of bits required by finite precision AUs to compute the SVD.

Then we turn our attention to the architectures which have been proposed to compute the SVD. Chapter 8 describes five such architectures. In Chapter 9 we analyze each of the five to determine their computation time for the SVD and the numbers of floating point or fixed point processors they require. Chapter 10 compares the resource requirements of the five architectures to determine which is best for VLSI implementation. Chapter 11 gives a similar comparison for three of the architectures constructed with CORDIC processors. Finally Chapter 12 gives the overall conclusions of the thesis.

2.0 COMPUTATION OF THE SVD

2.1 SVD Algorithms

The SVD of a real m-by-n matrix A [$A \in \Re^{m \times n}$] of rank r is given by

$$A = U \Sigma V^T \tag{2.1.1}$$

where

$$U \in \Re^{m \times m} \text{ and } U^T U = I_m$$
$$V \in \Re^{n \times n} \text{ and } V^T V = I_n$$
$$\Sigma \in \Re^{m \times n} \text{ and } \Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

and

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0, \quad \sigma_{r+1} = \ldots = \sigma_n = 0$$

The standard method for computing the SVD on mainframe computers is the Golub-Reinsch algorithm [Gol65, Gol70]. This algorithm computes the SVD in two stages. First the A matrix is reduced to upper bidiagonal form by multiplying it by a series of Householder matrices. The bidiagonal matrix is then reduced to a diagonal matrix (Σ) by iterative application of the symmetric QR algorithm (see [Gol83], Section 8.3 for details). This algorithm is very efficient in the single processor setting since it computes the SVD in $O(mn^2)$ floating point operations (flops) [Gol83]. However, attempts to map the Golub-Reinsch algorithm onto systolic architectures have failed because of its two-step nature and communication requirements [Fin83].

There are two other known methods for computing the SVD which are amenable to parallel processing. The first is the one-sided orthogonalization procedure of Hestenes [Hes58]. The second is a two-sided Jacobi algorithm due to Forsythe and Henrici [For60]. Both of these algorithms have been successfully adapted to systolic architectures.

The serial version of the Hestenes algorithm appears in Figure 2.1.1. The method consists of generating an orthogonal matrix V such that the columns of the matrix $H = AV$ are mutually orthogonal. The nonzero columns of H are then normalized to give $H = U\Sigma$ where $U^TU = I$ and Σ is a diagonal matrix. The SVD of A is easily seen to be $HV^T = U\Sigma V^T$. The V ma· ix is generated as a product of plane rotations each of which force orthogonality between two columns of A. The details of the algorithm are described in [Fin83, Luk80, Mor85].

The Jacobi algorithm appears in Figure 2.1.2. This iterative method uses a sequence of plane rotations to annihilate off-diagonal elements of A. The method is described in [Bre85a]. It is designed to operate on a square matrix. Rectangular matrices can be handled by adding columns of zeros to the original matrix or by performing a QR decomposition of the matrix and applying the Jacobi algorithm to R [Luk86, Bre85a].

2.2 Rotation Ordering Schemes

The feature of both the Hestenes and Jacobi algorithms which allows them to be implemented in parallel processors is that more than one of the rotations can be computed and applied simultaneously. In fact, for a matrix of n columns, n/2 rotations can be applied at the same time (if n is even). The difficult part of designing systolic architectures for the SVD is to insure that all n(n-1)/2 possible rotations are generated during each sweep while maintaining a nearest neighbor interconnection pattern. Two specific rotation orderings have been found which meet these restrictions. They will be described in the Hestenes algorithm context but they are equally applicable to the Jacobi algorithm.

```
V = I
for j = 1, 2, ..., n
      $\rho_j = a_j^T a_j$
for sweep = 1, 2, ...
      for i = 1, 2, ..., n
            for j = i+1, ..., n
                  $\gamma = a_i^T a_j$
                  $\psi = \frac{\rho_j - \rho_i}{2\gamma}$
                  $t = \frac{\mathrm{sign}(\psi)}{|\psi| + \sqrt{1 + \psi^2}}$
                  $\cos\theta = \frac{1}{\sqrt{1 + t^2}}$
                  $\sin\theta = t\cos\theta$
                  for k = 1, 2, ..., m
                        $\begin{bmatrix} a_{ki} & a_{kj} \\ v_{ki} & v_{kj} \end{bmatrix} = \begin{bmatrix} a_{ki} & a_{kj} \\ v_{ki} & v_{kj} \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$
                  $\begin{bmatrix} \rho_i \\ \rho_j \end{bmatrix} = \begin{bmatrix} \cos^2\theta & \sin^2\theta \\ \sin^2\theta & \cos^2\theta \end{bmatrix} \begin{bmatrix} \rho_i \\ \rho_j \end{bmatrix} + \begin{bmatrix} -2\gamma\cos\theta\sin\theta \\ 2\gamma\cos\theta\sin\theta \end{bmatrix}$
      end {for i and j}
end {for sweep}
for j = 1, 2, ..., n
      $\sigma_j = \sqrt{a_j^T a_j}$
      for i = 1, ..., m
            $u_{ij} = a_{ij} / \sigma_j$
end {for j}
```

Figure 2.1.1: The Hestenes SVD algorithm (with norm updating)

```
V = I
U = I
for sweep = 1, 2, ...
    for i = 1, 2, ..., n
        for j = i+1, ..., n
            $w = a_{ii}$
            $x = a_{ij}$
            $y = a_{ji}$
            $z = a_{jj}$
            $\psi_1 = \frac{z - w}{y + x}$
            $t_1 = \frac{\mathrm{sign}(\psi_1)}{|\psi_1| + \sqrt{1 + \psi_1^2}}$
            $\chi_1 = \frac{1}{\sqrt{1 + t_1^2}}$
            $\sigma_1 = \chi_1 t_1$
            $\psi_2 = \frac{z + w}{y - x}$
            $t_2 = \frac{\mathrm{sign}(\psi_2)}{|\psi_2| + \sqrt{1 + \psi_2^2}}$
```

Figure 2.1.2: The Jacobi SVD algorithm

(with Forsythe, Henrici procedure for rotation computations)

```
                $\chi_2 = \dfrac{1}{\sqrt{1 + t_2^2}}$
                $\sigma_2 = \chi_2 t_2$
                $c_1 = \chi_1\chi_2 + \sigma_1\sigma_2$
                $s_1 = \sigma_1\chi_2 - \chi_1\sigma_2$
                $c_2 = \chi_1\chi_2 - \sigma_1\sigma_2$
                $s_1 = \sigma_1\chi_2 + \chi_1\sigma_2$
                for k = 1, 2, ..., n
                    $\begin{bmatrix} a_{ik} & u_{ik} \\ a_{jk} & u_{jk} \end{bmatrix} = \begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} a_{ik} & u_{ik} \\ a_{jk} & u_{jk} \end{bmatrix}$
                    $\begin{bmatrix} a_{ki} & a_{kj} \\ v_{ki} & v_{kj} \end{bmatrix} = \begin{bmatrix} a_{ki} & a_{kj} \\ v_{ki} & v_{kj} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}$
                end {for k}
            end {for j}
        end {for i}
    end {for sweep}
```

Figure 2.1.2 (continued): The Jacobi SVD algorithm

(with Forsythe, Henrici procedure for rotation computations)

The first is the "odd-even" ordering of [Ste85]. In this scheme the rotation of odd column pairs is alternated with the rotation of even pairs. For example for n=8 the sequence of rotation pairs would be as follows:

Step							
1	(1,2)		(3,4)		(5,6)		(7,8)
2	2	(1,4)		(3,6)		(5,8)	7
3	(2,4)		(1,6)		(3,8)		(5,7)
4	4	(2,6)		(1,8)		(3,7)	5
5	(4,6)		(2,8)		(1,7)		(3,5)
6	6	(4,8)		(2,7)		(1,5)	3
7	(6,8)		(4,7)		(2,5)		(1,3)
8	8	(6,7)		(4,5)		(2,3)	1
9	(8,7)		(6,5)		(4,3)		(2,1)

Note that after n steps all n(n-1)/2 column pairs are generated by the odd-even ordering.

The second ordering scheme which is applicable is the "round robin" method of Brent and Luk [Bre85b]. This ordering is based on permuting the columns of A in a round robin sequence. For example for n = 8 the sequence of rotation pairs generated by this method is:

Step				
1	(1,2)	(3,4)	(5,6)	(7,8)
2	(1,4)	(2,6)	(3,8)	(5,7)
3	(1,6)	(4,8)	(2,7)	(3,5)
4	(1,8)	(6,7)	(4,5)	(2,3)
5	(1,7)	(8,5)	(6,3)	(4,2)
6	(1,5)	(7,3)	(8,2)	(6,4)
7	(1,3)	(5,2)	(7,4)	(8,6)
8	(1,2)	(3,4)	(5,6)	(7,8)

In this case all possible column pairs are generated in n-1 steps.

2.3 Input Data Matrices

2.3.1 Quantized Data Values

Many of the potential applications of the SVD in signal processing involve the decomposition of matrices of data values that have been quantized to a fixed number of bits. In most cases these values are generated by analog-to-digital conversion hardware. It is also frequently the case that the binary numbers used to represent the values contain an even number of 8-bit bytes in order to conform to hardware standards. Eight and 16 bits are the most common sizes of data words with 24 bits used less frequently where extra precision or dynamic range is needed.

For example, there are many applications of the SVD for digital image processing. In most cases the image is represented by a square array of 8-bit unsigned integers. (8-bit data allows 256 shades of grey which is usually more than adequate.) The SVD is also useful for processing data from arrays of sensors, such as seismometers and hydrophones. In these cases, the data values are normally quantized to signed, 16-bit values.

The observation that the original data matrices usually contain quantized values is crucial to the use of finite precision arithmetic to compute the SVD. One of the characteristics of quantized data is that the values contain quantization error. As a result when we compute the SVD of a quantized data matrix the singular values and singular vectors will also contain "quantization error." The effect of this quantization error is most apparent for the small singular values. In particular, we will show that any singular value with magnitude on the order of the "average" magnitude of the quantization error is highly questionable and cannot be used reliably in further computations. Therefore, the arithmetic used to compute the SVD need only insure that the magnitude of the "round-off" error is less than the "average" magnitude of the quantization error.

2.3.2 Scaling of the Input Data

For much of the development which follows it is convenient to assume that the elements of the data matrix (a_{ij}) satisfy $|a_{ij}| < 1$. Even though this is seldom the case in practice, we can very easily scale the data so that it is true by finding a constant (c) such that $|ca_{ij}| < 1$ for all i, j. After the SVD of the scaled values is computed, the original scale of the singular values can be reconstructed by dividing each of the computed values by c. In the fixed point context, such a scaling can be accomplished by shifts of the data values or by simply defining the fixed point data words so that the binary point is to the left of the most significant bit.

2.3.3 Input Data Representation

We will assume that the elements of the A matrix have been quantized by rounding the exact values to b+1 bits where the extra bit is used for sign representation. We will also assume that the matrix has been scaled so that $|a_{ij}| < 1$ for all i, j. Accordingly we can say that the quantization error (e_{ij}) associated with a_{ij} satisfies

$$|e_{ij}| \leq \frac{2^{-b}}{2} = 2^{-(b+1)} \equiv \varepsilon \qquad (2.3.3.1)$$

Further, we will assume that the quantization errors are uniformly distributed on the interval $[-\varepsilon,\varepsilon]$. With this assumption we see that the expected value of the quantization error is zero and its variance (s_b^2) is given by [Opp75]

$$s_b^2 = \frac{\varepsilon^2}{3} = \frac{2^{-b}}{12} \qquad (2.3.3.2)$$

Finally we will assume that all of the quantization errors in a matrix are independent.

2.4 Use of CORDIC Processors

The fundamental operations in both the Hestenes and Jacobi algorithms are the computation and application of plane rotations. It is easy to apply rotations using multipliers and adders. However, the computation of the rotation parameters (cos θ and sin θ) is difficult since the formulas involve square roots and divisions. An alternative form of arithmetic is available which is perfectly suited for the role of computing and applying rotations. The alternative is CORDIC arithmetic.

CORDIC is the acronym for a Coordinate Rotation Digital Computer developed by Volder for the calculation of trigonometric and hyperbolic functions [Vol59]. Further development and unifying mathematics for the CORDIC algorithm were done by Walther [Wal71] and Ahmed [Ahm81]. Recently, Cavallaro and Luk have presented two articles on CORDIC based SVD processors [Cav86, Cav87]. The basis for the CORDIC algorithm is coordinate rotation in a linear, circular, or hyperbolic coordinate system [Wal71]. The coordinate rotations are performed very quickly by a series of shift and add operations which can be implemented in very simple hardware. Of particular interest for the SVD algorithm is the ability of a CORDIC processing unit to compute a rotation angle in a single operation. A CORDIC unit can also apply a rotation to a pair of matrix elements in one operation.

The basic idea of the CORDIC algorithm is to compute or apply a rotation in incremental steps. For example, assume that we want to rotate the vector $[x \ \ y]^T$ through an angle θ to give the updated vector $[x' \ \ y']^T$. That is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.4.1}$$

If $\theta = \theta_0 + \theta_1 + \ldots + \theta_k$, we can apply the rotation incrementally

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix} \cdots \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 \\ \sin\theta_1 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} \cos\theta_0 & -\sin\theta_0 \\ \sin\theta_0 & \cos\theta_0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.4.2}$$

or pulling out a $\cos\theta_i$ term from each rotation matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \left(\prod_{i=0}^{k} \cos\theta_i\right) \begin{bmatrix} 1 & -\tan\theta_k \\ \tan\theta_k & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & -\tan\theta_0 \\ \tan\theta_0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.4.3}$$

The critical observation of the CORDIC algorithm is that the θ_i can be selected so that the $\tan\theta_i$ terms are powers of two. In the "standard" implementation of the CORDIC algorithm $\tan\theta_i = \delta_i 2^{-i}$ where $\delta_i = \pm 1$. The δ_i are selected so that

$$\theta \approx \sum_{i=0}^{k} \delta_i \tan^{-1}(2^{-i}) \tag{2.4.4}$$

If θ is in the range [-99°, 99°] δ_i can be found so that equation 2.4.4 is satisfied. The Hestenes and Jacobi algorithms use angles in the range [-90°, 90°].

If we choose the incremental angles so that $\tan\theta_i = \delta_i 2^{-i}$ we see that the matrix vector multiplications shown in equation 2.4.3 break down to a series of steps of the form

$$\begin{aligned} x_{i+1} &= x_i - \delta_i 2^{-i} y_i \\ y_{i+1} &= y_i + \delta_i 2^{-i} x_i \end{aligned} \qquad i = 0, 1, \ldots, k \tag{2.4.5}$$

where $x_0 = x$ and $y_0 = y$. The beauty of these steps is that the multiplications by 2^{-i} can be replaced by simple shifts . Therefore the majority of the rotation operation can be performed with shifters and adders. To complete the rotation we compute

$$x' = C\,x_{k+1} \text{ and } y' = C\,y_{k+1} \tag{2.4.6}$$

where C is given by

$$C = \prod_{i=0}^{k} \cos\theta_i = \prod_{i=0}^{k} \cos[\delta_i \tan^{-1}(2^{-i})] = \prod_{i=0}^{k} (1 + 2^{-2i})^{-\frac{1}{2}} \tag{2.4.7}$$

C is known as the CORDIC constant since for $k \geq 10$ it is approximately constant with a value of 0.607253. The full algorithm for applying a CORDIC rotation is shown in Figure 2.4.1.

Not only can we apply rotations with CORDIC processors but we can compute rotation angles as well. Often in the SVD algorithms we want to compute an angle which will rotate a vector $[x \;\; y]^T$ to $[r \;\; 0]^T$ where $r = (x^2 + y^2)^{1/2}$. The computation of angles needed to "annihilate" matrix elements is very easy with CORDIC processors. The algorithm for doing so is also shown in Figure 2.4.1.

We see that CORDIC processors are potentially very useful in SVD arrays. We will analyze the impact of using CORDIC processors in SVD architectures in later chapters.

Applying a rotation of angle θ to the vector $[x \;\; y]^T$

```
$x_0 = x$
$y_0 = y$
$\theta_0 = \theta$
for i = 0, 1, ..., k
    $\delta_i = \mathrm{sign}(\theta_i)$
    $x_{i+1} = x_i - \delta_i 2^{-i} y_i$
    $y_{i+1} = y_i + \delta_i 2^{-i} x_i$
    $\theta_{i+1} = \theta_i - \delta_i \tan^{-1}(2^{-i})$
$x' = C * x_{k+1}$
$y' = C * y_{k+1}$
```

Computing Angle $\theta \in [-90°, 90]$ to annihilate y in the vector $[x \;\; y]^T$

```
if $(x \geq 0)$
    $x_0 = x$
    $y_0 = y$
else
    $x_0 = -x$
    $y_0 = -y$
$\theta_0 = 0$
for i = 0, 1, ..., k
    $\delta_i = \mathrm{sign}(y_i)$
    $x_{i+1} = x_i + \delta_i 2^{-i} y_i$
    $y_{i+1} = y_i - \delta_i 2^{-i} x_i$
    $\theta_{i+1} = \theta_i + \delta_i \tan^{-1}(2^{-i})$
$\theta = \theta_{k+1}$
```

Figure 2.4.1: CORDIC algorithms for applying and computing plane rotations

3.0 REQUIREMENTS FOR FINITE PRECISION AUs IN SVD ARRAYS

In this section we will derive requirements for the arithmetic units used in SVD arrays.

3.1 Overflow Protection

With finite precision arithmetic we must insure that computed values do not overflow the maximum representable value. In the Jacobi SVD computation we need to be concerned about overflow since the diagonal elements of the updated A matrix grow. In the Hestenes algorithm we must be concerned about the growth in the elements of A but more importantly we must contend with the computation of inner products. In both algorithms, overflow is of no concern in the computation of U and V. Both of these matrices are orthogonal, so we are guaranteed that their elements satisfy $|u_{ij}| \leq 1$, $|v_{ij}| \leq 1$, for all i, j. Therefore we only need to analyze growth in the A matrix and its column norms.

By using matrix norm properties we can bound the size of the elements of the updated A matrix. Specifically we know that [Gol83]

$$\max_{(i,j)} |a_{ij}^{(k)}| \leq ||A^{(k)}||_2 \tag{3.1.1}$$

where $A^{(k)}$ is the updated matrix after k sweeps of either the Jacobi or Hestenes SVD algorithm. Since $A^{(k)}$ is obtained by multiplying $A^{(k-1)}$ by orthogonal matrices we know that [Gol83]

$$||A^{(k)}||_2 = ||A^{(k-1)}||_2 = \dots = ||A||_2 \tag{3.1.2}$$

As a result we can say that the elements of the updated A matrix are always bounded by $||A||_2$. In turn we can bound $||A||_2$ by using the relationship [Gol83]

$$\|A\|_2 \le \sqrt{mn} \max_{(i,j)} |a_{ij}| \tag{3.1.3}$$

By the definition of the data matrix given in Chapter 2 we know that $|a_{ij}| < 1$ for all (i,j). Therefore, we can conclude that

$$\max_{(i,j)} |a_{ij}^{(k)}| \le \|A\|_2 \le \sqrt{mn}\,, \ k = 1, 2, 3, \ldots \tag{3.1.4}$$

This relationship shows that if we want to prevent overflow in the computation of the updated m-by-n A matrix we must allocate $(\log_2 m + \log_2 n)/2$ bits to the left of the binary point in fixed point arithmetic. For the n-by-n matrices used in Jacobi arrays, we must allocate $\log_2 n$ bits.

In the Hestenes Algorithm, shown in Figure 2.1.1, we see that we must compute *inner products* prior to the computation of rotation parameters and at the end of the algorithm to compute the singular values. To determine the number of bits needed for overflow protection we must bound the size of the inner products. Using a series of norm relationships from [Gol83] we can see that the inner product of columns i and j of A ($a_i^T a_j$) must satisfy

$$|a_i^T a_j| \le \max_{(i,j)} |(A^T A)_{ij}| \le \|A^T A\|_2 = \|A\|_2^2 \le mn(\max_{(i,j)} |a_{ij}|)^2 \tag{3.1.5}$$

Again using the fact that $|a_{ij}| < 1$ we see that

$$|a_i^T a_j| < mn \tag{3.1.6}$$

So for the Hestenes algorithm we must add $\log_2(mn)$ bits to the left of the binary point in fixed point arithmetic to prevent overflow. If m = n, we must add 2 $\log_2 n$ bits. Thus, the Hestenes algorithm can require twice as many bits as the Jacobi algorithm to prevent overflow.

An alternate method of preventing overflow is to scale all elements of A so that $||A||_2 \leq 1$. From equations 3.1.4 and 3.1.5 given above we see that if we do so we can be sure that none of the computations in the SVD will overflow. The easiest way to insure that $||A||_2$ is bounded by 1 is to insure that the Frobenius norm of A ($||A||_F$) is bounded by 1 since $||A||_2 \leq ||A||_F$. By definition

$$||A||_F = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2 \tag{3.1.7}$$

To insure that $||A||_F \leq 1$ we could either divide all a_{ij} by $||A||_F$ or, since $|a_{ij}| < 1$, we could divide them all by $(mn)^{1/2}$.

Note that scaling the matrix in this way has the very desirable side effect of reducing the number of bits needed for overflow protection in the Hestenes algorithm. If we divide all elements by $(mn)^{1/2}$, in effect we have we have allocated $\log_2(mn)/2$ bits for overflow protection. This is half the number of bits needed to prevent overflow in the inner-product computation if we do not scale A. This reduction is achieved by taking advantage of the "squaring" effect of the inner-product computation. Since all elements are scaled down by $1/\sqrt{(mn)}$, the product of any two of them will be scaled down by $1/mn$. If $m = n$ we will need only n bits for overflow protection for the Hestenes algorithm, the same as in the Jacobi case.

In all further analysis we will assume that the A matrix has been prescaled so that $||A||_F \leq 1$.

3.2 Accurate Representation of Small Singular Values

We now turn our attention to the precision needed to represent small data values in the SVD output accurately . At first thought it would appear that we need as much precision as possible since the singular values can be arbitrarily small if the original matrix is nearly singular. However, we will show in this section that because we are dealing with matrices of quantized data values it is unnecessary to compute the SVD values to a precision less than the quantization error. The reason for this is that the potential variability of the elements of the SVD of a quantized data matrix is on the order of the quantization error.

3.2.1 Relationships Between the SVD and the Symmetric Eigenproblem

In order to quantify the variability of the singular values and vectors of a quantized data matrix we will draw on several results from symmetric eigenvalue theory. To utilize these results, we will rely on two relationships between the SVD and the real Schur decomposition of a symmetric matrix. The Schur decomposition is given by the following theorem.

Theorem 3.1 [Gol83] For a symmetric matrix $A \in \Re^{nxn}$ there exists an orthogonal Q such that

$$Q^T AQ = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_n) \qquad (3.2.1.1)$$

Proof See Golub and Van Loan [Gol83] theorem 8.1-1.

Based on the real Schur theorem we can establish the following two theorems.

Theorem 3.2 If $A \in \Re^{nxn}$ is symmetric and its real Schur decomposition is given by $Q^T AQ = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ where $Q = [q_1, q_2, \dots, q_n]$ is orthogonal, then the SVD of A is given by

$$U^T AV = \Sigma = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_n) \qquad (3.2.1.2)$$

where $\sigma_i = |\lambda_i|$, $V = Q$ and $U = [u_1, u_2, \dots, u_n]$ where $u_i = \text{sign}(\lambda_i)q_i$.

<u>Proof</u> Since Q is orthogonal V is orthogonal. Since the columns of U may differ only by sign from the columns of Q, U is also orthogonal. Finally

$$\Sigma_{ij} = u_i^T A v_j = \text{sign}(\lambda_j)\, q_i^T A q_j = 0, \; i \neq j$$

$$\Sigma_{ii} = \sigma_i = \text{sign}(\lambda_i)\, \lambda_i = |\lambda_i| \tag{3.2.1.3}$$

<u>Theorem 3.3</u> [Gol83] Given $A \in \Re^{m\text{x}n}$ $(m \geq n)$ with SVD given by $U^TAV = \Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$, if we form

$$B = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \tag{3.2.1.4}$$

then the eigenvalues of B are $\sigma_1, \sigma_2, \ldots, \sigma_n, -\sigma_1, -\sigma_2, \ldots, -\sigma_n$ and m - n 0's.

<u>Proof</u> Let an orthogonal matrix Q be defined by

$$Q = \frac{1}{\sqrt{2}} \begin{bmatrix} V & V & 0 \\ U_1 & -U_1 & \sqrt{2}\, U_2 \end{bmatrix} \tag{3.2.1.5}$$

where U_1 is the first n columns of U and U_2 is the last m-n columns of U. We see that

$$Q^TBQ = D = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n, -\sigma_1, -\sigma_2, \ldots, -\sigma_n, 0, \ldots, 0) \tag{3.2.1.6}$$

Since Q is orthogonal, Q^TBQ is a similarity transformation, so the diagonal elements of D are the eigenvalues of B.

These two theorems give us the connections which will allow us to apply results from the symmetric eigenproblem to the singular value problem.

3.2.2 Perturbation Bounds for the SVD of Quantized Data Matrices

There has been much work done on establishing bounds on the movement of eigenvalues and singular values caused by perturbation of the original data matrix. For example Wilkinson [Wil65] proves the following:

Theorem 3.4: Given two matrices A and B which have elements which satisfy $|a_{ij}| < 1$, $|b_{ij}| < 1$ for all i, j, if λ_i is a simple eigenvalue of A with corresponding eigenvector x_i then the matrix $A + \varepsilon B$ has a corresponding eigenvalue and eigenvector $[\lambda_i(\varepsilon), x_i(\varepsilon)]$ which satisfy

$$|\lambda_i(\varepsilon) - \lambda_i| = O(\varepsilon) \text{ and } |x_i(\varepsilon) - x_i| = O(\varepsilon) \qquad (3.2.2.1)$$

Proof See pages 65-67 of [Wil65].

Wilkinson also shows that for multiple eigenvalues the bound on the magnitude of the shift in the eigenvalues will involve fractional powers of ε (see page 70 of [Wil65]). For $\varepsilon < 1$, fractional powers of ε will be greater than ε. This means that simple eigenvalues will exhibit the smallest change in value as a result of perturbations to the original matrices.

Using Theorem 3.2, we can immediately apply Wilkinson's results to the SVD of a real symmetric matrix to show that the singular values and singular vectors of a perturbed matrix will be witnin $O(\varepsilon)$ of the unperturbed values. We can use Theorem 3.3 to show that such a relationship exists for the singular values of a general m x n matrix. That is $|\sigma(\varepsilon) - \sigma| = O(\varepsilon)$. In the particular case we are considering, the perturbation to the data matrix is caused by the quantization of the data values and ε is the maximum size of the quantization error. Therefore, the magnitude of the "error" in the singular values and vectors of a quantized data matrix is on the order of the quantization error. The following theorems quantify the perturbation bound for the singular values.

Theorem 3.5 [Gol83] Given A and $A + E \in \Re^{m\times n}$ $(m \geq n)$, then for $k = 1, 2, \ldots, n$

$$|\sigma_k(A+E) - \sigma_k(A)| \leq \sigma_1(E) = ||E||_2 \qquad (3.2.2.2)$$

Proof See page 286 of [Gol83].

Corollary 3.6 Given that E is a matrix of quantization error with $|e_{ij}| \leq \varepsilon = 2^{-(b+1)}$

$$|\sigma_k(A+E) - \sigma_k(A)| \leq \sqrt{mn}\,\varepsilon = \sqrt{mn}\,2^{-(b+1)} \qquad (3.2.2.3)$$

Proof From norm theory we know that $||E||_2 < ||E||_F$. For the given E the Frobenius norm can be bounded as follows.

$$||E||_F = \left[\sum_{i=1}^{m}\sum_{j=1}^{n} e_{ij}^2\right]^{1/2} \leq \left[\sum_{i=1}^{m}\sum_{j=1}^{n} \varepsilon^2\right]^{1/2} = \sqrt{mn}\,\varepsilon = \sqrt{mn}\,2^{-(b+1)} \qquad (3.2.2.4)$$

The corollary is obtained by substituting the bound for $||E||_2$ into Theorem 3.5.

Corollary 3.6 shows that if A is a square (n-by-n) matrix of quantized data values, the error in its singular values could be as high as n times the quantization error. This result is very significant, especially for small singular values, since the size of the bound is much larger than bounds based on machine precision arguments. For example, Golub and Van Loan show that for floating point arithmetic with precision μ, the error in the computation of the singular values of A is $O(\mu||A||_2)$ (see [Gol83] page 175). Therefore if $||A||_2 \approx 1$ and we are using 32 bit floating point arithmetic so that $\mu \approx 10^{-7}$, we expect to be able to compute singular values of order 10^{-7}. However if A is a 100 x 100 matrix of values quantized to 16 bits, the error bound given by Corollary 3.6 is of order 10^{-3}. Singular values with lower magnitude could be seriously corrupted by quantization error.

3.2.3 Variance of the SVD of a Quantized Data Matrix

While the bounds given in section 3.2.2 give us an indication of how large the error in the SVD of a quantized matrix can be, they do not give us any information on the "expected" value of the error. To establish requirements on the arithmetic used in the computation of the SVD, we need to know the size of the average error. Then we can design the arithmetic units so that their cumulative round-off error is smaller than the average quantization error. The following theorem gives an expression for the variance of the quantization error for the symmetric eigenvalue problem.

Theorem 3.7 [Vom83]: Let A be a real, symmetric n-by-n matrix which has simple eigenvalues $\lambda_1 < \lambda_2 < \cdots < \lambda_n$. Let $Q = [q_1, q_2, \ldots, q_n]$ be an orthogonal matrix such that $Q^T A Q = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$. Let A be perturbed by a symmetric matrix B whose elements are independent, random variables uniformly distributed on the range $[-\varepsilon, \varepsilon]$. If the eigenvalues of (A + B) are denoted by $\mu_1 < \mu_2 < \cdots < \mu_n$ then the variance of μ_i, $[s^2(\mu_i)]$ is given up to terms of the fourth order by

$$s^2(\mu_i) = s_b^2(2 - C_i) + O(s_b^4), \quad i = 1, 2, \ldots, n \tag{3.2.3.1}$$

where

$$C_i = \sum_{j=1}^{n} q_{ij}^4 \tag{3.2.3.2}$$

$$s_b^2 = \frac{\varepsilon^2}{3} \tag{3.2.3.3}$$

Proof See page 71 of [Vom83].

<u>Corollary 3.8</u> Given the conditions and notation of theorem 3.7 and ignoring terms greater than second order, the variance of the eigenvalues satisfy

$$s_b^2 \le s^2(\mu_i) \le 2s_b^2 \,, \; i = 1, 2, \ldots, n \tag{3.2.3.4}$$

<u>Proof</u> Since Q is orthogonal each of its elements must satisfy $|q_{ij}| \le 1$. Therefore $0 \le (q_{ij})^4 \le (q_{ij})^2$ and

$$C_i = \sum_{j=1}^{n} q_{ij}^4 \le \sum_{j=1}^{n} q_{ij}^2 = 1 \tag{3.2.3.5}$$

So $0 < C_i \le 1$. The corollary result is obtained by substituting these inequalities into the expression for $s^2(\mu_i)$ given in Theorem 3.7.

Corollary 3.8 shows that the variance of an eigenvalue of a real, symmetric, quantized data matrix is between one and two times the variance of the quantization noise. Using Theorem 3.2 we can draw the same conclusion for the singular values. However, the corollary is only applicable to the very restricted case of a real, symmetric matrix with simple eigenvalues. The simple eigenvalue restriction is not severe since multiple eigenvalues are highly unlikely with finite precision arithmetic and quantized data values. However, we do not expect to be dealing with symmetric matrices in algorithms involving the SVD. Therefore we would like some idea of the impact of removing the symmetry condition from Theorem 3.7 and Corollary 3.8.

The development given in [Vom83] for Theorem 3.7 shows that if we ignore higher order terms

$$s^2(\mu_i) = E(d_{ii}^2) \tag{3.2.3.6}$$

where E(x) is the expected value of x and $D = Q^T BQ$. We will use this observation to estimate the variance of the singular values of a non-symmetric matrix of quantized values.

Let us assume that A is not symmetric and that its SVD is given by $U^TAV = \Sigma$. We will also assume that A is perturbed by a non-symmetric matrix B whose elements are independent, random variables uniformly distributed on the range $[-\varepsilon,\varepsilon]$. We will define the singular values of A + B to be $\tau_1, \tau_2, \dots, \tau_n$. Using the observation given above for the symmetric case, we would expect the variance of τ_i $[s^2(\tau_i)]$ to be approximated by

$$s^2(\tau_i) = E(d_{ii}^2) \tag{3.2.3.7}$$

where $D = U^TBV$.

From the definition of D we see that

$$d_{ii} = \sum_{j=1}^{n}\sum_{k=1}^{n} u_{ji} v_{ki} b_{jk} \tag{3.2.3.8}$$

Therefore

$$E(d_{ii}^2) = \sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} u_{ji} u_{li} v_{ki} v_{mi} E(b_{jk} b_{lm}) \tag{3.2.3.9}$$

From the definition of B we can show that

$$E(b_{jk} b_{lm}) = \begin{cases} s_b^2, & \text{if } j = l \text{ and } k = m \\ 0, & \text{otherwise} \end{cases} \tag{3.2.3.10}$$

Therefore

$$E(d_{ii}^2) = s_b^2 \sum_{j=1}^{n}\sum_{k=1}^{n} u_{ji}^2 v_{ki}^2 = s_b^2 \sum_{j=1}^{n} u_{ji}^2 \sum_{k=1}^{n} v_{ki}^2 \tag{3.2.3.11}$$

Each of the sums in the last expression equals 1. Thus, we see that

$$s^2(\tau_i) = E(d_{ii}^2) = s_b^2 \qquad (3.2.3.12)$$

Equation 3.2.3.12 shows that the variance of the singular values of a real, non-symmetric quantized data matrix is equal to the variance of the quantization noise.

3.2.4 Experimental Verification of the Variance of the Singular Values of a Quantized Data Matrix.

The theoretical results of Section 3.2.3 were verified by computer simulation. In the simulation, a series of random A matrices, both symmetric and non-symmetric, were generated and their singular values computed. Then each A matrix was perturbed with a large number of B matrices where each b_{ij} was a uniformly distributed random variable on the range $[2^{-(b+1)}, 2^{-(b+1)}]$. The SVD was computed for each A + B matrix and the error of the singular values was computed. The simulation was performed for n = 4 and for b ranging from 3 to 23. Each A matrix was perturbed by a total of 200 B matrices. The results appear in Tables 3.2.4.1 and 3.2.4.2.

Table 3.2.4.1 shows the results for symmetric A matrices. Since A is symmetric its singular values are essentially its eigenvalues. As shown in equation 3.2.3.4 we expect the variance $s^2(\mu_i)$ of the eigenvalues to satisfy $s_b^2 \leq s^2(\mu_i) \leq 2s_b^2$. For the matrices in the simulation $s_b^2 = 2^{-2b}/12$. Table 3.2.4.1 shows the theoretical bounds and the minimum, mean and maximum experimental variances for each value of b. The experimental values are all within the theoretical bounds.

Table 3.2.4.2 shows the results for A non-symmetric. In this case we expect the variance of the singular values to be equal to s_b^2. The table shows

Table 3.2.4.1

Variance of the Singular Values of Symmetric Quantized Matrices

b	s_b^2	$2\ s_b^2$	min $s^2(\mu_i)$	mean $s^2(\mu_i)$	max $s^2(\mu_i)$
3	1.30e-03	2.60e-03	6.74e-04	1.51e-03	2.04e-03
5	8.14e-05	1.63e-04	1.14e-04	1.23e-04	1.37e-04
7	5.09e-06	1.02e-05	6.60e-06	7.53e-06	8.40e-06
9	3.18e-07	6.36e-07	4.48e-07	5.42e-07	5.82e-07
11	1.99e-08	3.97e-08	2.08e-08	2.57e-08	2.94e-08
13	1.24e-09	2.48e-09	1.74e-09	2.02e-09	2.27e-09
15	7.76e-11	1.55e-10	1.12e-10	1.29e-10	1.43e-10
17	4.85e-12	9.70e-12	6.47e-12	7.37e-12	8.36e-12
19	3.03e-13	6.06e-13	4.28e-13	4.72e-13	5.36e-13
21	1.89e-14	3.79e-14	2.15e-14	2.84e-14	3.47e-14
23	1.18e-15	2.37e-15	1.55e-15	1.82e-15	2.17e-15

Table 3.2.4.2

Variance of the Singular Values of Non-Symmetric Quantized Matrices

		A Non-singular			A-singular
b	s_b^2	min $s^2(\mu_i)$	mean $s^2(\mu_i)$	max $s^2(\mu_i)$	mean $s^2(\mu_i)$
3	1.30e-03	1.21e-03	1.32e-03	1.41e-03	1.03e-03
5	8.14e-05	7.54e-05	8.75e-05	9.46e-05	7.21e-05
7	5.09e-06	4.60e-06	4.97e-06	5.51e-06	4.15e-06
9	3.18e-07	3.11e-07	3.28e-07	3.37e-07	2.57e-07
11	1.99e-08	1.75e-08	2.06e-08	2.45e-08	1.59e-08
13	1.24e-09	1.16e-09	1.26e-09	1.38e-09	1.04e-09
15	7.76e-11	7.08e-11	7.57e-11	8.36e-11	6.60e-11
17	4.85e-12	4.29e-12	4.81e-12	5.14e-12	3.97e-12
19	3.03e-13	2.40e-13	2.84e-13	3.40e-13	2.49e-13
21	1.89e-14	1.60e-14	1.84e-14	2.04e-14	1.48e-14
23	1.18e-15	1.16e-15	1.24e-15	1.36e-15	1.06e-15

the minimum, mean and maximum variances. If the mean column is compared to the theoretical variance we see very good agreement over the full range of b.

Also displayed in Table 3.2.4.2 is a column for A non-symmetric and singular. The values in this column are the mean variances of the singular values when the original A matrix is singular. This experimental run was included to determine if singularity would have any significant impact on the stated results. The column shows that the variances for the singular matrices are consistently smaller than for the non-singular case but the difference is not large.

We conclude on the basis of the theoretical and experimental analysis that the variance of the singular values of a quantized data matrix is equal to the variance of the quantization error. We will use this result in Chapter 7 to compute the number of bits needed in the arithmetic units of SVD architectures. However the result is also significant by itself. It shows that we must be aware of the the characteristics of the original data in deciding how to use the results of an SVD computation. For example if the application of interest employs the smallest singular value as a control variable for follow-on actions, then we must be very wary of using singular values which are smaller than, say, two times the standard deviation of the quantization error. If our application uses only "significant" singular values and vectors, then our criterion for significance must incorporate the inherent precision of the data as well as the precision of the arithmetic used to compute the SVD.

4.0 THEORETICAL ERROR BOUNDS FOR THE JACOBI SVD ALGORITHM

In the previous section we computed the variance of the singular values of a quantized data matrix. Now we want to bound the size of the round off error for SVD algorithms as a function of the number of bits in the arithmetic units. This will allow us to set the number of bits in the AUs to insure that the round-off error is no larger than the quantization error. In this chapter we analyze the round-off error for the Jacobi SVD algorithm. The Hestenes algorithm is analyzed in Chapter 6.

4.1 Summary of Wilkinson's Error Analysis of the Jacobi Algorithm

Wilkinson [Wil65] performed a detailed analysis of the round-off error for the Jacobi algorithm for the symmetric eigenvalue problem. His analysis is based on the use of standard floating point or fixed point arithmetic with a machine precision of 2^{-t}. Wilkinson shows [see Wil65 pp. 279-281] that for symmetric $A \in \Re^{n\times n}$ with true eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, the Jacobi algorithm will compute eigenvalues $\mu_1, \mu_2, \ldots, \mu_n$ which satisfy

$$\left[\frac{\sum(\mu_i - \lambda_i)^2}{\sum \lambda_i^2}\right]^{1/2} \leq 18\, s\, n^{3/2} 2^{-t} (1 + 9\times 2^{-t})^{s(4n-7)}, \text{ for floating point} \tag{4.1.1}$$

$$\left[\frac{\sum(\mu_i - \lambda_i)^2}{\sum \lambda_i^2}\right]^{1/2} \leq s\, n^2\, 2^{-t} [(2n)^{1/2} + 5.84], \text{ for fixed point} \tag{4.1.2}$$

where s is the number of sweeps required by the Jacobi algorithm to reach convergence. The formula for fixed point arithmetic applies only if

$$||A||_F < 1 - \text{(maximum possible accumulated round-off error)} \qquad (4.1.3)$$

This condition insures that the computations will not overflow. We expect the accumulated round-off error to be << 1 so the condition of equation 4.1.3 is essentially satisfied if $||A||_F \leq 1$. We have already assumed in section 3.1 that the input matrix is scaled so that $||A||_F \leq 1$.

In the floating point error formula given above, the quantity 9×2^{-t} represents the error of the cosines and sines computed in the Jacobi algorithm [Wil65 pg 275]. The formulas used by Wilkinson to compute the cosines and sines are not the same as those shown in Figure 2.1.2. A cursory analysis indicates that the floating point errors for these formulas are slightly higher since they involve more floating point operations than those used by Wilkinson. We estimate the errors to be in the range of 12×2^{-t}. We will use that value from here on.

In the fixed point error formula, the factor 5.84 inside the square brackets represents the effect of the error in the computation of the rotation parameters. Again this factor will be highly dependent on the formulas used in the computation of the rotation parameters. Also note that this factor becomes insignificant as n grows large since the term $(2n)^{1/2}$ will dominate. We will simply ignore this factor from here on.

Finally in the floating point formula, the term $(1 + 9 \times 2^{-t})^{s(4n-7)}$ looks very ominous since it grows exponentially with n. However we expect $2^{-t} << 1$. Therefore the term $(1 + 9 \times 2^{-t})^{s(4n-7)}$ is approximately equal to one for all practical cases. Using this approximation and the fact that $\Sigma(\lambda_i)^2 = ||A||_F^2$, we see that Wilkinson's error bounds can be simplified to the following:

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \leq 24\,s\,n^{3/2}\,2^{-t}\,\|A\|_F, \text{ for floating point} \qquad (4.1.4)$$

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \leq \sqrt{2}\,s\,n^{5/2}2^{-t}\,\|A\|_F, \text{ for fixed point} \qquad (4.1.5)$$

Note that since the A matrix has been scaled so that $\|A\|_F \leq 1$, we could drop the norm term from the right hand side of these equations and the inequalities would still hold.

A careful comparison of our fixed point error formula to that given in [Wil65] will show that our bound has an extra factor of 2. This is because Wilkinson assumed that the matrix multiplications would be done with double length fixed point arithmetic. We are going to assume that single length fixed point arithmetic is used. Wilkinson shows that the use of single length fixed point math will result in a factor of two increase in the error bound for the Jacobi algorithm. This is because the error incurred in performing the multiply accumulate operation in a plane rotation will be doubled.

While Wilkinson states that these error bounds are for the symmetric eigenvalue problem, when a careful analysis is made of his derivation we find that it is equally applicable to the SVD of a general square matrix. Wilkinson's bounds are based on the multiplication of the matrix A on the right and left by a series of plane rotations. Nowhere in his derivation does he use the fact that A is symmetric. While he assumes that the rotations on the right will be the same as those on the left, this assumption is not necessary for his results to hold. Therefore, the error bounds shown above are directly applicable to the singular values of a general, real, square A.

4.2 Arithmetic Errors in Fixed Point CORDIC Arithmetic Units

We can use Wilkinson's fixed point error results to bound the round-off error in a Jacobi SVD architecture which uses fixed point CORDIC AUs. To do so we must develop expressions for the error of CORDIC operations. We will develop the expressions by taking advantage of the results obtained by Hu concerning quantization effects in CORDIC processors [Hu86]. While Hu's paper gives error expressions for the circular, linear, and hyperbolic modes of CORDIC operation, we will only use his results for the circular case.

Hu shows that there are two types of errors in the CORDIC algorithm. The first is the rounding error due to finite precision arithmetic. The second source of error is the finite number of iterations used in the CORDIC algorithm. For example if we wish to compute $\theta = \tan^{-1}(y/x)$, the CORDIC algorithm uses y as a control variable and tries to reduce it to zero. If we are applying a rotation, θ is used as the control variable and it is reduced to 0. However, after k iterations of the CORDIC algorithm, the value of the control variable will not be exactly 0. As a result, the output variables will not be exactly correct either. Hu calls this the approximation error. We prefer to use the term truncation error since it is caused by the fact that the CORDIC algorithm is truncated after a finite number of steps. The following theorems from [Hu86] give bounds for the two types of errors for the computation of angles and the application of rotations for fixed point CORDIC AUs.

4.2.1 Truncation Errors for Fixed Point CORDIC AUs

Theorem 4.1 - Truncation error for the computation of θ: Given initial values x and y, we wish to compute the angle required to rotate $[x \quad y]^T$ to $[x^* \quad 0]^T$ where $x^* = \text{sqrt}(x^2 + y^2)$. If we let θ^* denote the exact rotation angle and If x(k), y(k) and

$\theta(k)$ represent the output of a CORDIC algorithm after k iterations then

$$e_{t\theta}(k) \equiv \theta(k) - \theta^* = \tan^{-1}\left[\frac{y(k)}{x(k)}\right] \qquad (4.2.1.1)$$

$$\frac{|x(k) - x^*|}{x^*} \le \frac{y(k)^2}{2\,x(k)^2} \qquad (4.2.1.2)$$

As it stands, this theorem is not particularly useful since we must know the final output values x(k), y(k) and $\theta(k)$ in order to compute the error bounds. We would like to have apriori bounds. We can obtain such apriori bounds by combining this theorem with the result given in [Wal71] that $|e_{t\theta}(k)| < 2^{-k+1}$. Using this bound and equation 4.2.1.1 we can say that

$$|\tan^{-1}\left[\frac{y(k)}{x(k)}\right]| < 2^{-k+1} \qquad (4.2.1.3)$$

Using the small angle approximation that $\tan^{-1}(\theta) \approx \theta$, we have

$$\left|\frac{y(k)}{x(k)}\right| < 2^{-k+1} \qquad (4.2.1.4)$$

Finally combining this bound with equation 4.2.1.2 we can conclude that

$$\frac{|x(k) - x^*|}{x^*} \le \frac{(2^{-k+1})^2}{2} = 2^{-2k+1} \qquad (4.2.1.5)$$

This last formula shows that the relative truncation error in x(k) is very low. If we normalize the data matrix so that $||A||_F < 1$, we know that $x^* < 1$. In this case, $|x(k) - x^*| < 2^{-2k+1}$ which shows that the absolute truncation error in x(k) will be very small as well.

Theorem 4.2 - Truncation error for the application of rotations: Given initial values x, y, and θ, let x^* and y^* denote the exact values obtained by applying an exact rotation of angle θ to x and y. Also let $r = \mathrm{sqrt}(x^2 + y^2)$. If x(k), y(k) and θ(k) represent the output of a CORDIC algorithm with x, y, and θ as an input then

$$\frac{|x(k) - x^*|}{r} \leq |\theta(k)| \qquad (4.2.1.6a)$$

$$\frac{|y(k) - y^*|}{r} \leq |\theta(k)| \qquad (4.2.1.6b)$$

We can quantify the bounds of theorem 4.2 by again using the fact that $|\theta(k) - \theta^*| < 2^{-k+1}$. In this case $\theta^* = 0$ so we can say that $|\theta(k)| < 2^{-k+1}$. Using this approximation we see that

$$e_{t\,x}(k) \equiv |x(k) - x^*| \leq r\,2^{-k+1} \qquad (4.2.1.7a)$$

$$e_{t\,y}(k) \equiv |y(k) - y^*| \leq r\,2^{-k+1} \qquad (4.2.1.7b)$$

If the data values have been normalized so that $r < 1.0$, the absolute truncation error in both x(k) and y(k) is less than 2^{-k+1}.

4.2.2 Round-off Errors for Fixed Point CORDIC AUs

Hu estimates the round-off error for the vector $v(k) = [x(k) \quad y(k)]^T$. His theorems give bounds on the error between the value of v(k) computed with infinite precision arithmetic and Q[v(k)] which is the value computed in finite precision arithmetic. His results are given in the following theorem.

Theorem 4.3 - Round-off error for the application of rotations:
If a CORDIC AU uses fixed point arithmetic with precision ε, the round-off error

$e_r(k) = [e_{rx}(k)\ \ e_{ry}(k)]^T = Q[v(k)] - v(k)$ satisfies

$$\|e_r(k)\|_2 \leq \sqrt{2}\,\varepsilon[1 + (k-1)C(k)] \tag{4.2.2.1}$$

where $C(k) = 1/\text{CORDIC constant}$ (≈ 1.65).

If we use (t+1)-bit, rounded numbers with magnitudes less than 1, then $\varepsilon = 2^{-(t+1)}$ and the fixed point error bound is given by

$$\|e_r(k)\|_2 \leq \sqrt{2}\ 2^{-(t+1)}[1 + (k-1)C(k)] \leq \frac{7}{6} k\, 2^{-t} \tag{4.2.2.2}$$

Since $e(k) = [e_{rx}(k)\ \ e_{ry}(k)]^T$, the magnitude of either component of e(k) must be less than or equal to $\|e_r(k)\|_2$. Accordingly the round-off errors for x(k) and y(k) satisfy

$$e_{rx}(k) \equiv |Q[x(k)] - x(k)| \leq \frac{7}{6} k\, 2^{-t} \tag{4.2.2.3a}$$

$$e_{ry}(k) \equiv |Q[y(k)] - y(k)| \leq \frac{7}{6} k\, 2^{-t} \tag{4.2.2.3b}$$

Note that these bounds are only applicable for $k \leq t$. In the normal CORDIC implementation, during iteration k the updated value of x is determined from the equation $x_{k+1} = x_k \pm 2^{-k}y_k$. If k is greater than t then the term $2^{-k}y_k$ would be less than smallest representable value in the (t+1)-bit number system. So it makes no sense to perform more than t iterations of the CORDIC algorithm.

Hu does not give a specific bound for the round-off error for the computation of θ(k). In the standard CORDIC implementation θ(k) is given by

$$\theta(k) = \sum_{i=0}^{k} \delta_i \tan^{-1}(2^{-i}) \tag{4.2.2.4}$$

where $\delta_i = \pm 1$. In fixed point arithmetic a series of additions and subtractions can be performed with no round-off error. However that assumes that the variables being summed can be represented exactly in the fixed point number system. The terms we are summing [$\tan^{-1}(2^{-i})$, $i = 0, \ldots, k$] must be precomputed and rounded to (t+1) bits. So each term has an error with magnitude less than $2^{-(t+1)}$. Accordingly the round-off error $e_{r\theta}(k) = Q[\theta(k)] - \theta(k)$ must satisfy

$$|e_{r\theta}(k)| \leq (k+1)\, 2^{-(t+1)} \tag{4.2.2.5}$$

Again this formula is only applicable for $k \leq t$.

4.2.3 Total Errors for Fixed Point CORDIC AUs

We can combine the results of sections 4.2.1 and 4.2.2 to develop expressions for the total error of the output values of a CORDIC arithmetic unit. Specifically if we let z represent either x, y or θ, then the total error in z [$e_z(k)$] is given by

$$e_z(k) = Q[z(k)] - z^* = Q[z(k)] - z(k) + z(k) - z^* = e_{rz}(k) + e_{tz}(k) \tag{4.2.3.1}$$

Using the fact that the magnitude of a sum is bounded by the sum of the magnitudes we see that

$$|e_z(k)| \leq |e_{rz}(k)| + |e_{tz}(k)| \tag{4.2.3.2}$$

We can use this formula and the expressions given in sections 4.2.1 and 4.2.2 for the truncation and round-off errors to establish the following bounds on the total errors of the CORDIC variables:

$$|e_\theta(k)| \leq |e_{r\theta}(k)| + |e_{t\theta}(k)| = (k+1)2^{-(t+1)} + 2^{-k+1} \tag{4.2.3.3a}$$

$$|e_x(k)| \leq |e_{rx}(k)| + |e_{tx}(k)| = \frac{7}{6} k\, 2^{-t} + 2^{-k+1} \tag{4.2.3.3b}$$

$$|e_y(k)| \leq |e_{ry}(k)| + |e_{ty}(k)| = \frac{7}{6} k\, 2^{-t} + 2^{-k+1} \qquad (4.2.3.3c)$$

These formulas are valid for $k \leq t$.

4.2.4 Minimization of CORDIC Total Error

The equations given in the last section for total CORDIC error show that the error for all output variables decreases as t increases. However as k increases, the truncation error terms decrease but the round-off error terms increase. This indicates that there may be an optimum value for k which gives the lowest total error.

We can find the optimum value for k by treating t as a constant, taking the derivative of the total error expressions with respect to k and setting the derivatives equal to 0. Taking derivatives we find that

$$\frac{d|e_\theta(k)|}{dk} = 2^{-(t+1)} - 2^{-k+1}\ln(2) \qquad (4.2.4.1a)$$

$$\frac{d|e_x(k)|}{dk} = \frac{d|e_y(k)|}{dk} = \frac{7}{6} 2^{-(t+1)} - 2^{-k+1}\ln(2) \qquad (4.2.4.1b)$$

Note that the second derivative of all the error formulas is $2^{-k+1}\ln^2(2)$ which is greater than 0 for all k. Therefore we are finding a minimum.

Setting the first derivatives equal to 0 gives

$$2^{-(t+1)} = 2^{-k+1}\ln(2), \text{ for minimum } |e_\theta(k)| \qquad (4.2.4.2a)$$

$$\frac{7}{6} 2^{-t} = 2^{-k+1}\ln(2), \text{ for minimum } |e_x(k)| \text{ and } |e_y(k)| \qquad (4.2.4.2b)$$

Solving these equations for k gives

$$k = t + 2 + \log_2[\ln(2)] \approx t + 1.5, \text{ for minimum } |e_\theta(k)| \qquad (4.2.4.3a)$$

$$k = t + 1 + \log_2[\ln(2)] - \log_2\left(\frac{7}{6}\right) \approx t + 0.25,$$

$$\text{for minimum } |e_x(k)| \text{ and } |e_y(k)| \qquad (4.2.4.3b)$$

Since both expressions show that the optimum value of k is greater than t and the error expressions are only defined for $k \le t$, we can not directly use the results of the minimization analysis. However, the analysis is still useful because it shows that the CORDIC total error decreases for all values of $k \le t$. Therefore the minimum error is attained with $k = t$. If we let $k = t$ in the total error expressions, we find that

$$|e_\theta(t)| = (t+1)\, 2^{-(t+1)} + 2^{-t+1} = \frac{(t+5)}{2} 2^{-t} \qquad (4.2.4.4a)$$

$$|e_x(t)| = |e_y(t)| = \frac{7}{6} t\, 2^{-t} + 2^{-t+1} = \left(\frac{7}{6} t + 2\right) 2^{-t} \qquad (4.2.4.4b)$$

4.3 Error Bounds for SVD Algorithms With CORDIC AUs

In this section we will combine the results of sections 4.1 and 4.2 to develop error bounds for the Jacobi SVD algorithm with fixed point CORDIC AUs.

In his development of the error bound for the fixed point Jacobi algorithm, Wilkinson [Wil65] shows that the bound is the product of the number of rotations needed to reach convergence ($\approx sn^2$) times the error per rotation ($||F||$). He also shows that $||F||$ is given by

$$||F|| = ||G|| + ||\delta R||\ ||A|| \qquad (4.3.1)$$

where $||G||$ is the error incurred in applying a rotation and $||\delta R||$ is the error from the computation of the rotation parameters.

For the fixed point CORDIC case, if we use the Frobenius norm we can say that

$$\|G\|_F = \sqrt{2n}\ |e_x(t)| \tag{4.3.2}$$

since there are 2n terms computed when a rotation is applied to A and each could have an error as large as $|e_x(t)|$.

The matrix δR is the difference between the rotation matrix for angle θ and the rotation matrix for angle $\theta + e_\theta(t)$. From this definition we can see that all of the elements of δR will be zero except for four elements which can be represented as the following 2-by-2 matrix

$$\begin{bmatrix} \cos\theta - \cos[\theta+e_\theta(t)] & \sin\theta - \sin[\theta+e_\theta(t)] \\ \sin[\theta+e_\theta(t)] - \sin\theta & \cos\theta - \cos[\theta+e_\theta(t)] \end{bmatrix}$$

Therefore $\|\delta R\|_F$ is just the *Frobenius norm of this 2-by-2 matrix.*

Using trigonometric identities we can say that

$$\cos\theta - \cos[\theta+e_\theta(t)] = 2\sin\left(\frac{2\theta+e_\theta(t)}{2}\right)\sin\left(\frac{e_\theta(t)}{2}\right) \tag{4.3.3}$$

Since we expect $e_\theta(t)$ to be << 1, we can use the small angle approximation $\sin[e_\theta(t)/2] \approx e_\theta(t)/2$. Therefore

$$\cos\theta - \cos[\theta+e_\theta(t)] \approx \sin\left(\frac{2\theta+e_\theta(t)}{2}\right) e_\theta(t) \tag{4.3.4}$$

Using the fact that the magnitude of sin is bounded by 1 we can say that

$$|\cos\theta - \cos[\theta+e_\theta(t)]| \leq |e_\theta(t)| \tag{4.3.5}$$

Using a similar development we can also say that

$$|\sin\theta - \sin[\theta + e_\theta(t)]| \le |e_\theta(t)| \tag{4.3.6}$$

Accordingly $||\delta R||_F$ is given by

$$||\delta R||_F \le \sqrt{4\,|e_\theta(t)|^2} = 2\,|e_\theta(t)| \tag{4.3.7}$$

Substituting the bounds for $||\delta R||_F$ and $||G||_F$ into the expression for $||F||$ given in equation 4.3.1 and assuming that we have constrained $||A||_F \le 1$, we see that

$$||F||_F \le \sqrt{2n}\;|e_x(t)| + 2\,|e_\theta(t)| \tag{4.3.8}$$

Substituting in the expressions for $|e_x(t)|$ and $|e_\theta(t)|$ from section 4.2, we see that

$$||F||_F \le [\sqrt{2n}\,(\frac{7}{6}t+2) + 2\,(\frac{t+5}{2})\,]\,2^{-t} \approx 2\sqrt{n}\;t\,2^{-t} \tag{4.3.9}$$

Finally, the error bound for the singular values of the Jacobi algorithm with fixed point CORDIC AUs is sn^2 times $||F||_F$ or

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \le 2\,s\,n^{5/2}\,t\,2^{-t},\ \text{for fixed point CORDIC AUs} \tag{4.3.10}$$

if $||A||_F$ is ≤ 1.

4.4 Summary of Theoretical Error Bounds for the Jacobi SVD Algorithm

In summary we have been able to show the following bounds for the errors of the singular values generated by the Jacobi algorithm:

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \leq 24\, s\, n^{3/2}\, 2^{-t}\, ||A||_F, \text{ for floating point} \qquad (4.4.1a)$$

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \leq \sqrt{2}\, s\, n^{5/2} 2^{-t}\, ||A||_F, \text{ for fixed point} \qquad (4.4.1b)$$

$$\left[\sum_{i=1}^{n}(\mu_i - \lambda_i)^2\right]^{1/2} \leq 2\, s\, n^{5/2}\, t\, 2^{-t}, \text{ for fixed point CORDIC} \qquad (4.4.1c)$$

These formulas apply only if the data matrix has been scaled so that $||A||_F \leq 1$.

If we assume that the computation errors are distributed uniformly to the n singular values, we can say that the computational error in the ith singular value $[e(\mu_i) = \mu_i - \lambda_i]$ satisfies

$$|e(\mu_i)| \leq 24\, s\, n\, 2^{-t}, \text{ for floating point arithmetic} \qquad (4.4.2a)$$

$$|e(\mu_i)| \leq \sqrt{2}\, s\, n^2\, 2^{-t}, \text{ for fixed point arithmetic} \qquad (4.4.2b)$$

$$|e(\mu_i)| \leq 2\, s\, n^2\, t\, 2^{-t}, \text{ for fixed point CORDIC arithmetic} \qquad (4.4.2c)$$

5.0 EXPERIMENTAL ERROR BOUNDS FOR THE JACOBI SVD ALGORITHM

A simulation was performed to test the theoretical bounds given in section 4.4. This chapter describes the simulation and its results.

5.1 Simulation Programs

Computer programs were developed to perform the Jacobi SVD using double precision, floating point arithmetic; t-bit, floating point arithmetic; t-bit, fixed point, CORDIC arithmetic; and t-bit, fixed point arithmetic. The programs were written in the C programming language and were run on a VAX 11/780 with a floating point accelerator.

The programs compute the SVD of a series of matrices with the Jacobi algorithm, using full double precision arithmetic. The VAX double precision floating point format has 64 bits with 53 bits allocated to the mantissa. This gives a machine precision of approximately 10^{-16}. Then the programs compute the SVD of the same matrices with the Jacobi algorithm using one of the t-bit arithmetics. Finally, the programs compute the errors in the singular values, compile appropriate statistics on the errors and print out the statistics and the theoretical bounds.

For each type of t-bit arithmetic two separate main programs were created and tested. The first computes the round-off error of the singular values as a function of the array size (n) while holding t constant (at t = 23). It does so for n varying from 10 to 50 in steps of 10. We wanted to test the routines at higher values of n but at n = 50 the processing time for a single matrix was on the order of 25 minutes. Since the computation time for the Jacobi algorithm is $O(n^3)$, the time to process a single matrix with n = 100 would be on the order of three and a half hours!

The second main routine computes the round-off error of the singular values as a function of the number of bits, t, while holding n constant at 20. The value of t was increased from 15 to 29 in steps of 2. The low limit of 15 was selected since this is the number of bits available in many signal processing chips available today. The upper limit of 29 is imposed by the functions used to simulate the t-bit arithmetic. All of these functions use integer variables at some point. In the VAX implementation of C, the longest available integer is 32 bits. We need 1 bit to represent the sign and we will see that we need 1 bit to allow for growth of values in some low level routines. This leaves 30 bits available for t. One additional bit was allowed as a safety margin for the simulation giving the upper limit of 29 for t.

The following is a brief description of the programs for each type of arithmetic.

5.1.1 Routines for the Double Precision Jacobi Algorithm

The specific version of the Jacobi algorithm used in the simulation appears in Figure 5.1.1.1. It uses the Forsythe-Henrici [For60] procedure for computing rotation parameters. The algorithm performs sweeps on the n-by-n matrix A until the sum of the squares of A's off-diagonal elements has been reduced to a specified threshold (delta) or until the number of sweeps exceeds a preset limit (maxsweeps). In order to save time, the U and V matrices are not computed. This version of the Jacobi SVD algorithm does not guarantee that the computed singular values will be positive or that they will be in any specific order. Therefore the main routines must take the absolute value of the diagonal elements of the output matrix and sort them prior to computing error statistics.

```
off = $\sum_{i=1}^{n}\sum_{\substack{j=1\\ i\neq j}}^{n} a_{ij}^2$
maxsweeps = 10
sweeps = 0
delta = 1.0e-12 * off
while (off > delta  and  sweeps ≤ maxsweeps) do
        for  i = 1, 2, ..., n
                for  j = i+1, ..., n
                        $w = a_{ii},\quad x = a_{ij},\quad y = a_{ji},\quad z = a_{jj}$
                        $r_1 = \frac{z - w}{y + x}$
                        $t_1 = \frac{\mathrm{sign}(r_1)}{|r_1| + \sqrt{1 + r_1^2}}$
                        $x_1 = \frac{1}{\sqrt{1 + t_1^2}}$
                        $r_1 = x_1 t_1$
                        $r_2 = \frac{z + w}{y - x}$
                        $t_2 = \frac{\mathrm{sign}(r_2)}{|r_2| + \sqrt{1 + r_2^2}}$
```

Figure 5.1.1.1: Jacobi SVD algorithm used in the simulation

```
                $x_2 = \frac{1}{\sqrt{1 + t_2^2}}$
                $r_2 = x_2 t_2$
                $c_1 := x_1 x_2 + r_1 r_2$
                $s_1 := r_1 x_2 - x_1 r_2$
                $c_2 := x_1 x_2 - r_1 r_2$
                $s_2 := r_1 x_2 + x_1 r_2$
                for k = 1, 2, ..., n
                    $\begin{bmatrix} a_{ik} \\ a_{jk} \end{bmatrix} = \begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} a_{ik} \\ a_{jk} \end{bmatrix}$
                    $\begin{bmatrix} a_{ki} & a_{kj} \end{bmatrix} = \begin{bmatrix} a_{ki} & a_{kj} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}$
                end { for k }
            end { for j }
        end { for i }
        $off = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} a_{ij}^2$
        sweeps = sweeps + 1
    end { while }
```

Figure 5.1.1.1 (continued): Jacobi SVD algorithm used in the simulation

5.1.2 Routines for t-bit Floating Point Arithmetic

The programs for the t-bit floating point simulation used exactly the same algorithm as the double precision version but all arithmetic operations (+, -, ×, ÷ and √) were performed with t-bit floating point numbers. The t-bit operations are based on the function "tround" which rounds the mantissa of a double precision number to t bits. The listing for tround is shown in Figure 5.1.2.1. Its operation is described in detail in the listing.

The function rounds a floating point number x so that

$$\left|\frac{\text{tround}(x) - x}{x}\right| \leq 2^{-t} \qquad (5.1.2.1)$$

which is the expected performance for t-bit floating point arithmetic. The operation of tround was verified by rounding a large number (10000) of random x's for each value of t and compiling statistics on the errors. Table 5.1.2.1 gives the results. For all values of t the maximum error encountered was within the theoretical bound (2^{-t}). If the errors were uniformly distributed on $[-2^{-t}, 2^{-t}]$ we would expect their variance to be $2^{-2t}/3$. The computed variances are all smaller. The reason they are smaller is that the errors are not uniformly distributed on the interval $[-2^{-t}, 2^{-t}]$. For example, if x is near the boundary between two exponent ranges then the relative error for tround(x) is in the range $[-2^{-(t+1)}, 2^{-t}]$. Therefore we expect the variance of the errors to be smaller than $2^{-2t}/3$.

The function tround was used as the basis for functions which perform t-bit addition, subtraction, multiplication, division and square root. The double precision Jacobi algorithm was modified to call these t-bit arithmetic functions to produce the t-bit floating point SVD algorithm.

```
double tround(x)
double x;
/*   This function rounds the mantissa of a double precision number to t bits
where t is less than 31.  It operates in the following manner. The input variable x
is copied into variable y.  Then y is decomposed into an integer exponent "expon"
and a double precision representation of its mantissa using the standard function
frexp.  The function frexp guarantees that the magnitude of the mantissa will lie in
the half closed interval [0.5, 1.0).  The absolute value of the mantissa is scaled
up by a factor of 2^t by loading the integer t into its exponent using the standard
function ldexp.  We now have a value whose magnitude is guaranteed to lie in
the range [2^(t-1), 2^t).  We want to round this value to t bits.  We do this by copying
the value into the long integer variable temp.  The VAX version of C truncates the
fractional portion of a number when converting from floating point to integer.
Therefore to produce a rounded integer we must add 0.5 to the double precision
number prior to the conversion.  After the conversion, the integer temp has the
scaled, absolute value of the mantissa of x rounded to t bits.  To produce the
output value we rescale the data by loading temp into the double precision
variable y and setting its exponent to (expon - t) using the function ldexp.  Then
depending on the original sign of x we return either y or -y.                    */
{ /*  begin tround */
      extern int t;        /*   Number of bits in mantissa of floating point words     */
      int expon;           /*   Exponent of input variable x                           */
      long int temp;       /*   Temporary storage                                      */
      double y;            /*   Workcopy of input variable x                           */
      y = x;
      temp = ldexp(frexp(fabs(y),&expon),t) + 0.5;
      y = ldexp((double)temp,(expon - t));
      if (x >= 0.0)
            return y;
      else
            return -y;
} /*  end tround */
```

Figure 5.1.2.1: Program listing for function tround

Table 5.1.2.1

Performance of Function tround

t	2^{-t}	max |error|	mean error	variance	$2^{-2t}/3$
4	6.2500e-02	5.8757e-02	-5.1309e-04	6.9444e-04	1.3021e-03
6	1.5625e-02	1.5381e-02	-8.9919e-05	4.4120e-05	8.1380e-05
8	3.9063e-03	3.8896e-03	1.9244e-05	2.7241e-06	5.0863e-06
10	9.7656e-04	9.6692e-04	-2.9220e-06	1.7171e-07	3.1789e-07
12	2.4414e-04	2.4200e-04	2.7826e-07	1.0857e-08	1.9868e-08
14	6.1035e-05	6.0011e-05	2.0881e-07	6.7621e-10	1.2418e-09
16	1.5259e-05	1.5036e-05	-4.0812e-08	4.1625e-11	7.7610e-11
18	3.8147e-06	3.7769e-06	-5.2880e-09	2.5845e-12	4.8506e-12
20	9.5367e-07	9.4863e-07	-3.8101e-09	1.6493e-13	3.0316e-13
22	2.3842e-07	2.3665e-07	1.4934e-09	1.0310e-14	1.8948e-14
24	5.9605e-08	5.8711e-08	-1.5083e-10	6.3130e-16	1.1842e-15
26	1.4901e-08	1.4833e-08	-4.9466e-11	3.9698e-17	7.4015e-17
28	3.7253e-09	3.6240e-09	-3.2055e-12	2.5160e-18	4.6259e-18
30	9.3132e-10	9.2277e-10	1.7780e-12	1.5844e-19	2.8912e-19

5.1.3 Routines for t-bit Fixed Point Arithmetic

The programs for t-bit fixed point arithmetic were very similar to those for t–bit floating point arithmetic. However there were two major differences.

First the rotation parameters were computed using the original double precision functions. The reason for this is that the Forsythe-Henrici formulas can not be easily implemented in fixed point arithmetic. The formulas have intermediate values whose magnitude can range from well above 1 to 0. To simulate the effect of t-bit rotation parameters, the sines and cosines produced by the double precision routines were rounded to t bits prior to their application. This departure from a full fixed point implementation is not significant for the simulation since the error caused by inaccurate rotation angles is overwhelmed by the error caused by the application of rotations. However, if one actually wanted to use fixed point AUs in an SVD array, the rotation parameter computations would have to be reformulated to prevent the wide range of values seen in the Forsythe-Henrici formulas. Alternatively the fixed point AUs could be used exclusively in the rotation application units with floating point AUs used in the rotation computation units.

The second change made to the t-bit floating point routines was the replacement of the function tround with the function "tfixround". This function allows the simulation of t-bit fixed point arithmetic even though the variables it operates on are double-precision floating point values. A listing of tfixround is shown in Figure 5.1.3.1. Its operation is described in detail in the listing.

The effect of the function is to round a value x so that

$$|\mathrm{tfixround}(x) - x| \le 2^{-(t+1)} \qquad (5.1.3.1)$$

which is the defining characteristic of fixed point arithmetic. Additionally tfixround

```
double tfixround(x)
double x;

/* This function allows the simulation of fixed point arithmetic by rounding double
precision numbers in the range [-1, 1] to the closest value which can be
represented by a t-bit fixed point number. It operates in the following manner. If
the magnitude of the input variable is less than the fixed point precision, 2^-(t+1)),
then the value 0 is returned. Otherwise the input variable x is copied into variable
y. Then the absolute value of y is decomposed into an integer exponent,
"expon", and a double precision representation of its mantissa using the standard
function frexp. The function frexp guarantees that the magnitude of "mantissa"
will lie in the half closed interval [0.5, 1.0). The mantissa is scaled up by a factor
of 2^(t+expon) by loading the integer (t + expon) into its exponent using the
standard function ldexp. We now have a value whose magnitude is guaranteed
to lie in the range [0.5, 2^t). We want to round this value to t bits. We do this by
converting the value into the long integer variable temp. The VAX version of C
truncates the fractional portion of a number when converting from a floating point
to integer. Therefore to produce a rounded integer we must add 0.5 to the
double precision number prior to the conversion. After the conversion, the
integer temp has the scaled, absolute value of the mantissa rounded to t bits. To
produce the output value we rescale the data by loading temp into the double
precision variable y and setting its exponent to (-t) using the function ldexp. Then
depending on the original sign of x we return either y or -y.              */

{ /* begin tfixround */
    extern int t;           /*..Number of bits in mantissa of floating point words */
    int expon;              /*  Exponent of input variable x                       */
    long int temp;          /*  Temporary storage                                  */
    double y;               /*  Work copy of input variable x                      */
    double  mantissa;       /*  Absolute value of mantissa of input variable x     */
    extern double precision; /* Precision of t-bit fixed point arithmetic = 2^-(t+1) */
    if (fabs(x) < precision)
        return 0.0;
    else
    {
        y = x;
        mantissa = frexp(fabs(y), &expon);
        temp = (long int)(ldexp(mantissa, t + expon) + 0.5);
        y = ldexp((double)temp, -t);
        if (x >= 0.0)
            return y;
        else
            return -y;
    }
} /* end tfixround */
```

Figure 5.1.3.1: Program listing for function tfixround

produces errors which are uniformly distributed on the range $[-2^{-(t+1)}, 2^{-(t+1)}]$. The characteristics of tfixround were verified in the same way as those of tround. Table 5.1.3.1 gives the performance of tfixround as a function of t. The table shows that the maximum errors are bounded by $2^{-(t+1)}$ and the variance of the errors is very close to the theoretical value $(2^{-2t}/12)$ for all values of t.

5.1.4 Routines for t-bit Fixed Point CORDIC Arithmetic

The main routines for t-bit CORDIC arithmetic used the double precision algorithm as a base but the computation and application of rotations were performed with CORDIC routines. All data-related computations used integer arithmetic. In order to perform the t-bit fixed point SVD, the initial data matrices were scaled by multiplying them by $2^t/||A||_F$ and rounded to integers so that only t bits are ever used.

The critical elements of the *CORDIC* implementation of the *Jacobi algorithm* are the routines CORDICangle and COApplyRotation which compute an angle and apply a rotation with CORDIC arithmetic. They are shown and described in Figure 5.1.4.1 and 5.1.4.2. They implement the routines shown in Figure 2.4.1 with a few exceptions.

The first exception is the representation of the angles. In the normal CORDIC implementation a rotation angle (θ) is computed using the formula

$$\theta = \sum_{i=0}^{t-1} \delta_i \tan^{-1}(2^{-i}) \qquad (5.1.4.1)$$

where $\delta_i = \pm 1$. In our application we found it convenient to multiply all the $\tan^{-1}(2^{-i})$ terms by $2^{(t-1)}$, round the results, and store them in the integer array phi. Our angles (θ_t) are computed using the formula

Table 5.1.3.1

Performance of Function tfixround

t	$2^{-(t+1)}$	max |error|	mean error	variance	$2^{-2t}/12$
3	6.2500e-02	6.2500e-02	-5.4687e-04	1.2913e-03	1.3021e-03
5	1.5625e-02	1.5625e-02	-9.0820e-05	8.0638e-05	8.1380e-05
7	3.9063e-03	3.9063e-03	-1.3831e-04	5.1322e-06	5.0863e-06
9	9.7656e-04	9.7656e-04	1.0052e-05	3.2373e-07	3.1789e-07
11	2.4414e-04	2.4414e-04	-2.7475e-06	2.0101e-08	1.9868e-08
13	6.1035e-05	6.0916e-05	-5.4458e-07	1.2448e-09	1.2418e-09
15	1.5259e-05	1.5229e-05	-1.7985e-07	7.9034e-11	7.7610e-11
17	3.8147e-06	3.8143e-06	1.2093e-08	4.6776e-12	4.8506e-12
19	9.5367e-07	9.5139e-07	-6.5040e-09	2.9478e-13	3.0316e-13
21	2.3842e-07	2.3840e-07	3.9663e-10	1.9058e-14	1.8948e-14
23	5.9605e-08	5.9415e-08	2.0060e-09	1.1632e-15	1.1842e-15
25	1.4901e-08	1.4834e-08	-1.9386e-10	7.5562e-17	7.4015e-17
27	3.7253e-09	3.7249e-09	8.5022e-11	4.7658e-18	4.6259e-18
29	9.3132e-10	9.3058e-10	-3.2347e-11	2.7733e-19	2.8912e-19

```
void CORDICangle(u, v, theta)
long int u, v, *theta;
/* This function computes the angle, theta, needed to rotate the vector [u  v]^T to
the vector [r  0]^T where r = ± sqrt(u^2 + v^2). If |v| is less than a preset threshold,
10, then theta is set to 0. Otherwise the function computes theta using the
CORDIC algorithm shown in Figure 2.4.1. The angle theta is returned as a
scaled, integer version of the true angle
                    theta = 2^(t-1) * theta in radians
where t is the number of bits being used in the CORDIC operations.        */
{ /* begin CORDICangle */
    int i;
    long int x, y, angle;
    if (abs(v) <= 10)
        *theta = 0;
    else
    {
        if (u >= 0)
        {
            x = u;
            y = v;
        }/* end if of if (u > = 0) */
        else
        {
            x = -u;
            y = -v;
        } /* end else of if (u >= 0) */
        angle = 0;
        for (i = 0; i < t; i++)
        {
            if (y > 0)
            {
                angle = angle + phi[i];
                u = shiftadd(x, y, i);
                v = shiftadd(y, -x, i);
            }/* end if of if (y > 0) */
            else
            {
                angle = angle - phi[i];
                u = shiftadd(x, -y, i);
                v = shiftadd(y, x, i);
            }/* end else of if (y > 0) */
            x = u;
            y = v;
        } /* end for (i = 0; i < t; i++) */
        *theta = angle;
    } /* end else of if (abs(v) <= 10) */
}
```

Figure 5.1.4.1: Program listing for function "CORDICangle"

```
void COApplyRotation(u, v, theta, C)
long int *u, *v;
long int theta;
double C;  /*  CORDIC constant */
/*  This function applies a rotation of angle theta to the vector [u  v]^T.  If theta = 0
on entry no action is taken so the original values of u and v are maintained.  If
theta ≠ 0 then the rotation is applied using the CORDIC algorithm shown in
Figure 2.4.1.                                                              */
{ /* begin COApplyRotation */
        int i;
        long int x, y, a, b;
        if (theta == 0)
                return;
        else
        {
                x = *u;
                y = *v;
                for (i = 0; i < t; i++)
                {
                        if (theta >= 0)
                        {
                                a = shiftadd(x, -y, i);
                                b = shiftadd(y, x, i);
                                theta = theta - phi[i];
                        }
                        else
                        {  /* theta < 0*/
                                a = shiftadd(x, y, i);
                                b = shiftadd(y, -x, i);
                                theta = theta + phi[i];
                        }
                        x = a;
                        y = b;
                }/*  end for (i = 0; i < t; i++) */
                *u = rint(C * (double)x);
                *v = rint(C * (double)y):
        } /*  end else of if (theta == 0)  */
} /*  end COApplyRotation */
```

Figure 5.1.4.2: Program listing for function "COApplyRotation"

$$\theta_t = \sum_{i=0}^{t-1} \delta_i \, phi[i] \approx 2^{(t-1)}\theta \qquad (5.1.4.2)$$

The resulting angle is an approximate, scaled version of the original angle. However, since we use the same phi[i] terms when we apply the rotation, we end up transferring the exact angle information.

The second exception is the inclusion of a threshold test for the computation of rotation angles. If the element to be annihilated is sufficiently small (10 was found to be a good threshold) the angle is set to zero. If the rotation application routine detects a zero angle it simply returns the original values. The reason the threshold was included was to allow a fair comparison between the CORDIC results and the fixed point and floating point results. In the floating point and fixed point ca es a threshold must be included in the rotation computation to prevent overflows. If the threshold is reached, the rotation parameters, cos θ and sin θ, are set to 1 and 0 respectively, i.e the angle is set to 0.

The third exception to the normal CORDIC implementation is the use of the function shiftadd to compute the updated data values. The true formula for computing the updated value of x in step i is

$$x_{i+1} = x_i \pm 2^{-i} y_i \qquad (5.1.4.3)$$

In practice this computation is performed by shifting y_i right by i bits and adding it to or subtracting it from x_i.

$$x_{i+1} = x_i \pm \underset{(i\,bits)}{shiftright}(y_i) \qquad (5.1.4.4)$$

However we discovered in the course of our simulation that this leads to very large errors in the CORDIC SVD algorithm. The reason for the large errors is that the shift operation truncates the scaled value of y so instead of computing x as in equation 5.1.4.3 you actually compute

$$x_{i+1} = x_i \pm \underset{\text{(to t bits)}}{\mathrm{truncate}}(2^{-i} y_i) \tag{5.1.4.5}$$

This truncation causes the errors in the CORDIC algorithm to be biased and add up very rapidly.

The function shiftadd shown in Figure 5.1.4.3 corrects this by computing

$$x_{i+1} = x_i \pm \underset{\text{(to t bits)}}{\mathrm{round}}(2^{-i} y_i) \tag{5.1.4.6}$$

By rounding the scaled value of y the bias is eliminated from the CORDIC error so the error accumulates much less rapidly. Note that the shiftadd function includes special code to handle the case when $2^{-i}y_i$ has a fractional part which is exactly equal to 0.5. The extra code insures that values that fall into this category are rounded to even. As described in [Was82] and [Yoh73] this procedure gives an unbiased rounding. It is crucial that unbiased rounding be used in the CORDIC algorithm. Since we are always "dividing" by powers of two it is quite common for values with fractions equal to 0.5 to appear. In fact we found in our simulations that such values appear in approximately 5% of the shiftadd operations. If biased rounding is used the errors found in the singular values produced by a CORDIC SVD algorithm are much higher than if unbiased rounding is used. Figure 5.1.4.4 shows the effect of different roundings on the SVD errors. Clearly unbiased rounding is best.

Unfortunately the arithmetic of the VAX11/780 gives biased rounded results. We used floating point arithmetic in the shiftadd function to simulate the unbiased rounding operation on the VAX. However in an actual implementation the shiftadd operation can be performed correctly with t-bit fixed point arithmetic augmented with three extra bits (see [Was82] for details).

```
long int shiftadd(x, y, i)
long int x, y;
int i;
/* This function returns an unbiased rounding of the value u = (x + 2^-i y). It
operates in the following manner. It computes the true double precision value of
u and copies the absolute value of u into v. It then truncates v to an integer and
stores the result in k. If the difference between v and k is not equal to 1/2 then
the function returns the value of u rounded to the closest integer with the
standard function "rint". In the Vax implementation of rint, if v-k = 0.5 rint will
round away from 0 which gives a biased rounding. To produce an unbiased
rounding, when v-k = 0.5 the value of k is tested to determine if it is odd. If so it is
incremented so that it is even. Then the value of k with the appropriate sign is
returned.                                                                        */

{ /* begin shiftadd */
    double u, v;
    double onehalf = 0.5;
    long int k;
    long int one = 1;
    u = (double)x + ((double)y * inv2[i])              /* inv2[i] = 2^-i     */
    v = fabs(u);
    k = (long int)v;
    if ((v - k) != onehalf)
        return rint(u)
    else
    { /* v - k = 0.5 so return to nearest even integer to u */
        if (k & one)
            k++;
        if (u >= 0.0)
            return k;
        else
            return -k;
    }
}
```

Figure 5.1.4.3: Program listing for function "shiftadd"

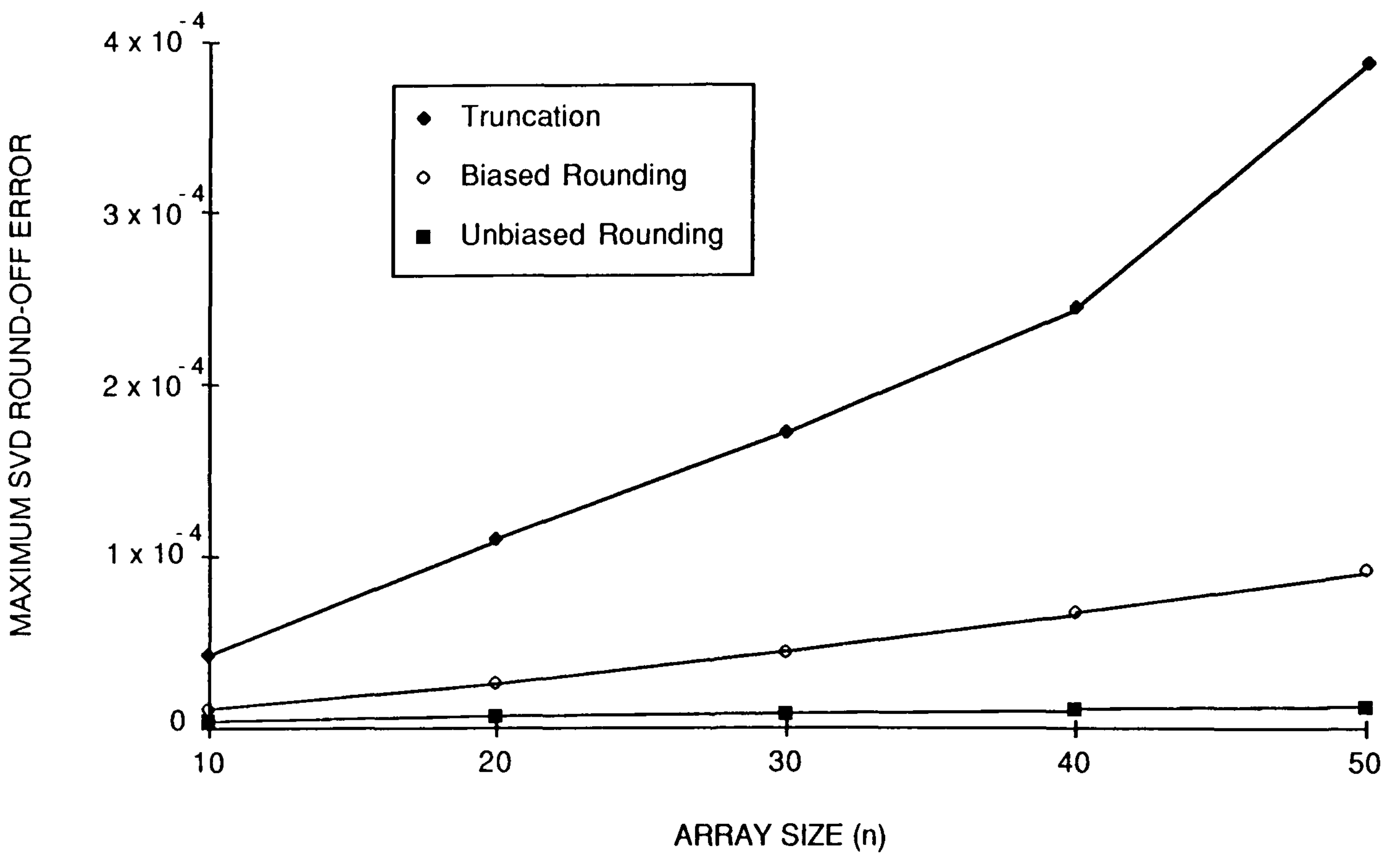


Figure 5.1.4.4: Impact of different rounding methods on the CORDIC SVD

One additional bit is needed in a CORDIC AU to allow for growth in the data values. In the routine COApplyRotation, when the input vector $[x, y]^T$ is rotated by angle theta its length grows. To return its length to the proper value we must multiply the rotated vector's coordinates by the CORDIC constant. Prior to the multiplication, the vector's coordinates are approximately 1.6 times their final values. Therefore, it is possible that the intermediate values in COApplyRotation could have magnitudes as larger as 1.6×2^t. This value requires t+1 bits to be represented accurately.

One additional observation must be made about the CORDIC constant. In our simulation we have used the expedient method of applying C with double precision arithmetic. In actual CORDIC hardware there are two approaches to applying C. The first was recommended in Haviland and Tuszynski [Hav80] and involves repeating some of the CORDIC iterations. The purpose of the repetition is to force C towards unity. However, this repetition of iterations has the undesirable side effects of increasing the round-off error and lengthening the CORDIC computation time. For example when $t = 24$, Haviland and Tuszynski show that a total of 12 iterations must be repeated to make $C \approx 1$. So we can expect the error and computation time to be 50% greater. To compensate for the increased error we would need additional bits in the CORDIC word. The alternative is to implement a special purpose multiplier that scales any input value by C. Such a dedicated hardware unit could apply the scale factor quickly and with little or no error. However, it would require a significant amount of chip area. We have chosen to simulate the dedicated multiplier approach since the 50% increase in time and round-off error caused by the repetition scheme seems prohibitively high.

5.2 Input Matrices for the Simulation

The input matrices for the Jacobi algorithm simulations were n-by-n matrices of random numbers which were (hopefully) uncorrelated and uniformly distributed on the range [-1, 1]. We used the VAX C-library routine "random" to generate the matrix elements.

5.2.1 Norms of Uniform Random Matrices

We have seen that the bounds for round-off errors of the singular values depend on the norm of the original matrices. The expected value of the norm of uniformly distributed random matrices is directly related to the number of elements in the matrices. The following analysis shows the dependency of $||A||_F$ and $||A||_2$ on n.

For the Frobenius norm

$$E(||A||_F^2) = E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} E(a_{ij}^2) = n^2 E(a_{ij}^2) = \frac{n^2}{3} \qquad (5.2.1.1)$$

We can make the final step in this equation because a_{ij} is uniformly distributed on the range [-1, 1] which implies that it has zero mean and variance 1/3. Since the $E(||A||_F{}^2) = n^2/3$ we can expect $||A||_F = n/\sqrt{3}$.

Due to the complex definition of the 2-norm, we can not compute its expected value directly. However from norm theory we know that $||A||_2 \geq ||A||_F/\sqrt{n}$ so we can say that $E(||A||_2) \geq E(||A||_F)/\sqrt{n}$ or

$$E(||A||_2) \geq \frac{n}{\sqrt{3}\sqrt{n}} = \frac{\sqrt{n}}{\sqrt{3}} \qquad (5.2.1.2)$$

We ran an experiment to verify these expected values for $||A||_F$ and $||A||_2$.

The experiment involved generating a series of random matrices (for $10 \leq n \leq 50$) and computing their norms. The results appear in Table 5.2.1.1 The table shows that the Frobenius norm is essentially equal to $n/\sqrt{3}$ over the full range of n. The experiment also shows that the $E(||A||_2) \approx \sqrt{n}$ which is certainly greater than $\sqrt{n}/\sqrt{3}$.

5.2.2. Effects of Normalizing the A matrix

In our simulations we normalized the A matrices to prevent overflows in the t-bit arithmetic computations. For the Jacobi algorithm, the ideal normalization procedure would be to divide all of A's elements by $||A||_2$. This would insure that the maximum value in the computations would always be equal to 1. Thus dividing by $||A||_2$ not only prevents overflows but also makes full use of the range of numbers available in t-bit arithmetic. However we do not know $||A||_2$ until the SVD computation is complete. Therefore we are forced to use alternative normalization constants such as $||A||_F$ or n. Both of these constants will prevent overflow and they are either known or can be computed easily prior to the start of the SVD. However, normalizing with either $||A||_F$ or n reduces the range of the data values. The following computations show the effect of normalizing with $||A||_F$ or n on $E(||A||_2)$ for uniformly distributed random matrices.

For $B = (1/||A||_F) * A$

$$E(||B||_2) = \frac{E(||A||_2)}{||A||_F} = \frac{\sqrt{n}}{n/\sqrt{3}} = \frac{\sqrt{3}}{\sqrt{n}} \qquad (5.2.2.1)$$

For $C = (1/n) * A$,

$$E(||C||_F) = \frac{E(||A||_F)}{n} = \frac{n/\sqrt{3}}{n} = \frac{1}{\sqrt{3}} \qquad (5.2.2.2)$$

Table 5.2.1.1

Norms of Matrices of Uniformly Distributed Random Numbers

A an n-by-n matrix with all $a_{ij} \in [-1, 1]$

n	$E(\|\|A\|\|_F)$	$n/\sqrt{3}$	$E(\|\|A\|\|_2)$	$\sqrt{n}$
10	5.780	5.774	3.239	3.162
20	11.499	11.547	4.764	4.472
30	17.377	17.321	5.854	5.477
40	23.029	23.094	6.748	6.325
50	29.044	28.868	7.923	7.071

$B = A/||A||_F$

n	$\|\|B\|\|_F$	$E(\|\|B\|\|_2)$	$\sqrt{3}/\sqrt{n}$
10	1.000	0.561	0.548
20	1.000	0.414	0.387
30	1.000	0.337	0.316
40	1.000	0.293	0.274
50	1.000	0.273	0.245

$C = A/n$

Theoretical $E(||C||_F) = 1/\sqrt{3} = 0.577$

n	$E(\|\|C\|\|_F)$	$E(\|\|C\|\|_2)$	$1/\sqrt{n}$
10	0.578	0.324	0.316
20	0.575	0.238	0.224
30	0.579	0.195	0.183
40	0.576	0.169	0.158
50	0.581	0.158	0.141

$$E(||C||_2) = \frac{E(||A||_2)}{n} = \frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}} \tag{5.2.2.3}$$

Note that we have used the experimental result that $E(||A||_2) \approx \sqrt{n}$ to derive these results. These theoretical values for the norms of the normalized matrices were confirmed by experiment. The experimental results are shown in Table 5.2.1.1.

The impact of normalizing by $||A||_F$ or n is to reduce the largest singular value to $O(1/\sqrt{n})$. If we have designed our fixed point arithmetic to handle numbers as large as 1, we are not using their full range. In fact we could reduce the number of bits allocated to overflow protection (guard bits) by $\log_2(\sqrt{n})$. Note also that there is only a slight penalty to pay for using n as the normalization constant instead of $||A||_F$. The $E(||C||_2)$ is only a factor of $\sqrt{3}$ less than $E(||B||_2)$ which translates into a penalty of less than 1 guard bit. At the same time using n as the normalization constant avoids the necessity of computing $||A||_F$.

The conclusions given above are really true only for uniform random matrices. However, they do give us some indication of what can be expected in real applications. Even though the results show that we could use fewer guard bits, it is probably unwise to do so. In real systems it is possible that sensor failures or extremely large signals will cause a continuous stream of maximum values to appear at the input of an SVD array. This case will cause the largest singular value to jump to 1 under any of the normalization methods we have discussed. In our computation of the number of bits needed in fixed point AUs (given in Chapter 7) we have assumed that the input matrices are normalized by dividing all elements by n.

5.3 Simulation Results

The results of our simulation of the Jacobi SVD algorithm with t-bit arithmetic are given in the form of three tables. Table 5.3.1 gives the results for t-bit floating point arithmetic. Table 5.3.2 shows the output for t-bit, fixed point, CORDIC arithmetic and Table 5.3.3 covers t-bit, fixed point arithmetic. Each table shows the minimum, mean and maximum round-off errors of the singular values as a function of n and t. The tables include a column labeled "s" which is the number of sweeps needed by the t-bit Jacobi algorithm to reach convergence. There is also a column labeled "#" which is the number of samples analyzed to produce the minimum, mean and maximum errors. Finally the tables have a column showing the theoretical bounds and a column giving the ratio of the bound to the maximum error. The theoretical bounds were computed using the following formulas which were computed in section 4.4

$$|e(\mu_i)| \leq 24\, s\, n\, 2^{-t}, \text{ for floating point arithmetic} \qquad (5.3.1a)$$

$$|e(\mu_i)| \leq \sqrt{2}\, s\, n^2\, 2^{-t}, \text{ for fixed point arithmetic} \qquad (5.3.1b)$$

$$|e(\mu_i)| \leq 2\, s\, n^2\, t\, 2^{-t}, \text{ for fixed point CORDIC arithmetic} \qquad (5.3.1c)$$

The tables show that the maximum round-off errors are well within the theoretical bounds for all three types of arithmetic. The tables also show that the bound to maximum error ratio grows rapidly as a function of n for all three types of arithmetic. This indicates that the relationship of the bound to n is too strong. We do not see strong trends in the bound to maximum ratio as a function of t for the floating point and fixed point cases. This indicates that the bound's functional dependence on t is appropriate. There is a strong upward trend of the ratio as a function of t for CORDIC arithmetic.

Table 5.3.1

Round-off Error of the Jacobi SVD Algorithm
With t-bit, Floating Point Arithmetic

$$\text{bound} = 24\, s\, n\, 2^{-t}$$

As a function of n (t = 23)

n	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
10	5	250	3.66e-09	8.38e-07	3.36e-06	1.43e-04	43
20	6	400	5.90e-10	1.60e-06	4.81e-06	3.43e-04	71
30	7	270	3.67e-08	2.25e-06	6.77e-06	6.01e-04	89
40	7	200	3.11e-09	2.77e-06	7.25e-06	8.01e-04	110
50	7	200	5.88e-09	3.35e-06	8.27e-06	1.00e-03	121

As a function of t (n = 20)

t	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
15	6	200	2.45e-06	4.53e-04	1.27e-03	8.79e-02	69
17	6	200	8.15e-07	1.14e-04	3.23e-04	2.20e-02	68
19	6	200	1.01e-07	2.80e-05	8.25e-05	5.49e-03	67
21	6	200	7.19e-09	6.89e-06	2.10e-05	1.37e-03	65
23	6	200	1.13e-08	1.65e-06	4.78e-06	3.43e-04	72
25	6	200	5.56e-09	3.67e-07	1.18e-06	8.58e-05	73
27	6	200	1.44e-10	9.70e-08	2.92e-07	2.15e-05	74
29	6	200	4.61e-11	1.71e-08	6.71e-08	5.36e-06	80

Table 5.3.2

Round-off Error of the Jacobi SVD Algorithm
With t-bit, Fixed Point, CORDIC Arithmetic

$$\text{bound} = 2\, s\, n^2\, t\, 2^{-t}$$

As a function of n (t = 23)

n	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
10	5	180	1.18e-09	9.51e-07	3.18e-06	2.74E-03	862
20	6	180	2.11e-08	2.27e-06	6.48e-06	1.32E-02	2030
30	6	150	3.07e-08	3.34e-06	8.25e-06	2.96E-02	3591
40	7	120	1.16e-08	4.53e-06	1.02e-05	6.14E-02	6035
50	7	100	5.09e-07	5.56e-06	1.21e-05	9.60E-02	7926

As a function of t (n = 20)

t	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
15	5	180	1.69e-08	4.03e-04	1.29e-03	1.83E+00	1421
17	5	200	1.79e-08	1.02e-04	2.91e-04	5.19E-01	1783
19	5	160	2.12e-07	2.73e-05	8.71e-05	1.45E-01	1664
21	6	180	3.32e-08	6.61e-06	2.15e-05	4.81E-02	2237
23	6	200	7.89e-08	2.29e-06	6.48e-06	1.32E-02	2030
25	6	200	2.96e-09	4.87e-07	1.32e-06	3.58E-03	2712
27	6	200	8.80e-11	1.45e-07	4.10e-07	9.66E-04	2353
29	6	180	1.23e-10	3.19e-08	8.39e-08	2.59E-04	3089

Table 5.3.3

Round-off Error of the Jacobi SVD Algorithm
With t-bit, Fixed Point Arithmetic

$$\text{bound} = \sqrt{2}\, s\, n^2\, 2^{-t}$$

As a function of n (t = 23)

n	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
10	5	250	1.05e-09	3.23e-07	1.08e-06	8.43e-05	78
20	6	400	9.92e-10	5.26e-07	2.03e-06	4.05e-04	199
30	7	240	4.41e-09	7.22e-07	3.06e-06	1.06e-03	347
40	7	200	5.40e-09	8.22e-07	2.61e-06	1.89e-03	724
50	7	200	3.84e-09	8.53e-07	3.12e-06	2.95e-03	945

As a function of t (n = 20)

t	s	#	min $\lvert e(\mu_i)\rvert$	mean $\lvert e(\mu_i)\rvert$	max $\lvert e(\mu_i)\rvert$	bound	bnd/max
15	4	160	1.06e-08	1.79e-04	5.10e-04	6.91e-02	135
17	5	180	3.29e-08	3.68e-05	1.16e-04	2.16e-02	186
19	5	180	2.36e-08	8.42e-06	2.65e-05	5.39e-03	204
21	5	200	9.40e-09	2.13e-06	7.66e-06	1.35e-03	176
23	5	200	3.16e-09	5.07e-07	2.01e-06	3.37e-04	167
25	5	180	1.63e-09	1.43e-07	5.53e-07	8.43e-05	152
27	5	120	3.82e-10	5.55e-08	1.29e-07	2.11e-05	163
29	6	160	2.67e-10	8.80e-09	3.19e-08	6.32e-06	198

These initial results from the simulation were somewhat disappointing since they showed ' at the bounds were excessively loose and had too strong a dependence on n. Our end objective is to determine the number of bits needed in the SVD AUs to insure that the round-off error is approximately the same magnitude as the quantization error. To do this we need tight bounds on the round-off error so that we do not assign too many bits to the task of controlling the round-off error. In order to improve the bounds we carefully observed the accumulation of errors in the singular values during the simulation runs. This observation allowed us to develop some approximate statistical bounds which are much tighter than the theoretical, worst case bounds. These statistical bounds are developed in the next section.

5.4 Statistical Bounds for Round-off Errors in the Jacobi Algorithm

5.4.1 Assumptions/Observations

Our statistical bounds are based on the following assumptions and observations:

a. We need not be concerned with the errors in computing the rotation angles in the error bounds. Van Loan has shown [Van85] that the angles can be computed very roughly and the Jacobi algorithm will still converge. As long as we maintain the orthogonality of the rotations and maintain some significant number of bits in the angles, the algorithm works. The primary impact of using approximate angles is to increase the number of sweeps. Therefore, the round-off error attributable to the angle computations is already accounted for by the "s" term in our bounds.

b. We will assume that we are operating on a non-singular matrix. This is not a crucial assumption but it makes the analysis easier. In all of our simulation runs the Jacobi algorithm converged nicely whether or not the matrix was singular. Further, there appears to be no significant difference in the magnitude of the round-off error of a nearly zero singular value and that of a normal singular value.

c. As the Jacobi algorithm progresses towards convergence the error in the diagonal elements grows with *n* and *s*. However, the error in the off-diagonal terms does not grow since each off-diagonal element is zeroed out during each sweep. This zeroing effectively resets the error in the off-diagonal terms to zero.

d. As the Jacobi algorithm progresses, the rotation angles converge to zero. The reason that the angles get smaller is that the diagonal elements are growing while the off-diagonal elements are getting smaller. The angles are computed by taking the inverse tangent of the ratio of the sum or difference of off-diagonal elements to the sum or difference of diagonal elements. Since the off-diagonal elements are going to zero while the diagonal elements grow, the ratios will go to zero as will the angles.

5.4.2 General Analysis of Statistical Bounds

When we combine these observations we come to an interesting conclusion that allows us to compute approximations for the round-off errors of the diagonal elements. The conclusion is that as the Jacobi algorithm progresses the round-off error in the off-diagonal elements has less and less impact on the round-off error of the diagonal elements. Let's take a look at the application of a rotation to two elements of the A matrix using fixed point arithmetic to show why this is true.

Assume that we are applying a rotation of angle θ whose cosine is c and whose sine is s to the elements a_{ii} and a_{ij}. Assume also that the matrix elements have accumulated error e_{ii} and e_{ij} to this point in the algorithm. That is, we have $\hat{a}_{ii} = a_{ii} + e_{ii}$ and $\hat{a}_{ij} = a_{ij} + e_{ij}$. The computation we wish to perform is

$$\begin{bmatrix} \hat{a}_{ii}^{(k)} \\ \hat{a}_{ij}^{(k)} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} \hat{a}_{ii}^{(k-1)} \\ \hat{a}_{ij}^{(k-1)} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{ii}^{(k-1)} + e_{ii}^{(k-1)} \\ a_{ij}^{(k-1)} + e_{ij}^{(k-1)} \end{bmatrix} \qquad (5.4.2.1)$$

Let us define the updated A elements to be the sum of a true value and an error component

$$\begin{bmatrix} \hat{a}_{ii}^{(k)} \\ \hat{a}_{ij}^{(k)} \end{bmatrix} = \begin{bmatrix} a_{ii}^{(k)} \\ a_{ij}^{(k)} \end{bmatrix} + \begin{bmatrix} e_{ii}^{(k)} \\ e_{ij}^{(k)} \end{bmatrix} \qquad (5.4.2.2)$$

Equating terms we see that

$$\begin{bmatrix} a_{ii}^{(k)} \\ a_{ij}^{(k)} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{ii}^{(k-1)} \\ a_{ij}^{(k-1)} \end{bmatrix} \qquad (5.4.2.3)$$

and

$$\begin{bmatrix} e_{ii}^{(k)} \\ e_{ij}^{(k)} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} e_{ii}^{(k-1)} \\ e_{ij}^{(k-1)} \end{bmatrix} + \begin{bmatrix} d_{ii}^{(k)} \\ d_{ij}^{(k)} \end{bmatrix} \qquad (5.4.2.4)$$

were the $d_{ii}^{(k)}$ and $d_{ij}^{(k)}$ terms are the round-off errors incurred in performing the multiplications and additions of the kth rotation computation. With this formulation we see that the only way that the overall magnitude of the round-off error grows in the Jacobi algorithm is through the effect of the d terms. Once errors have occurred they can only be redistributed from element to element.

The redistribution of errors is of concern only if the relatively large round-off errors in the diagonal elements could be redistributed to the off-diagonal elements by one rotation and then redistributed to other diagonal elements by subsequent rotations. If this scenario were to occur then the round-off errors in the diagonal elements could grow very rapidly. Observations c and d given above above eliminate this possibility.

The primary way that round-off error is redistributed from off-diagonal to diagonal elements or vice versa is if the angle is large so that its sine is large. In the initial sweeps we may have large angles so we may have some redistribution of round off errors. But the size of the errors being redistributed is small. As the number of sweeps grows the angles get smaller. As they do, the redistribution of errors decreases. With the cosine going to ±1 and the sine going to 0, equation 5.4.2.4 shows that the errors tend to stay where they are. That is

$$\begin{bmatrix} e_{ii}^{(k)} \\ e_{ij}^{(k)} \end{bmatrix} \approx \begin{bmatrix} e_{ii}^{(k-1)} \\ e_{ij}^{(k-1)} \end{bmatrix} + \begin{bmatrix} d_{ii}^{(k)} \\ d_{ij}^{(k)} \end{bmatrix} \qquad (5.4.2.5)$$

Additionally, since the error of the off-diagonal elements are small in relation to the error in the diagonal elements, their impact on the diagonal error would be small even if they were totally redistributed. That is, if the rotation worked perfectly so that $\hat{a}_{ij}$ was reduced to zero then, because of the norm preservation property of the rotation, we would have

$$e_{ii}^{(k)} = \left[(e_{ii}^{(k-1)})^2 + (e_{ij}^{(k-1)})^2 \right]^{1/2} + d_{ii}^{(k)} \qquad (5.4.2.6)$$

Since $e_{ij}^{(k-1)}$ is small relative to $e_{ii}^{(k-1)}$, it will have little impact in this equation. Therefore we can say that

$$e_{ii}^{(k)} \approx e_{ii}^{(k-1)} + d_{ii}^{(k)} \tag{5.4.2.7}$$

Based on the observations given above we are going to make the assumption that the total round-off error for a diagonal element after N rotations have been applied to it is given by

$$e_{ii}^{(N)} \approx \sum_{k=1}^{N} d_{ii}^{(k)} \tag{5.4.2.8}$$

We will further assume that the d_{ii} are independent random variables with zero mean and variance s_t^2.

With these assumptions we see that $e_{ii}^{(N)}$ is the sum of N independent identically distributed random variables. The central limit theorem tells us that $e_{ii}^{(N)}$ will be a Gaussian random variable. Since the d_{ii} terms are zero mean, $e_{ii}^{(N)}$ will also be zero mean. The variance of $e_{ii}^{(N)}$ will be N times the variance of the individual errors. Since $e_{ii}^{(N)}$ is a Gaussian random variable, there is a 99% chance that its magnitude will lie within 3 standard deviations of its mean. Hence we expect

$$|e_{ii}^{(N)}| \leq 3\sqrt{N}\, s_t \tag{5.4.2.9}$$

In the Jacobi algorithm each element of the A matrix is involved in 2(n-1) rotations during each sweep. If it takes s sweeps to reach convergence, then N = 2s(n-1) at convergence. Accordingly we can say that the total round-off error for a diagonal element (e_{ii}), which is the round off error of singular value μ_i, is bounded by

$$|e_{ii}| = |e(\mu_i)| \leq 3\sqrt{2\,s\,(n-1)}\; s_t \leq \sqrt{18\,s\,n}\; s_t \tag{5.4.2.10}$$

This shows that the error in the singular values should be O[√(ns)] which is a much lower dependence on n than that shown by the bounds developed in Chapter 4. To determine the proportionality constant and the dependence on t we must find expressions for s_t, the standard deviation of the error of a multiply accumulate operation. This standard deviation is dependent on the type of arithmetic used.

5.4.3 Statistical Bound for t-bit Fixed Point Arithmetic

For t-bit fixed point arithmetic the multiply accumulate operation has a round-off error which is the sum of 2 independent errors each of which are uniformly distributed on the range $[-2^{-(t+1)}, 2^{-(t+1)}]$. Accordingly

$$s_t = 2^{-t}/\sqrt{6} \tag{5.4.3.1}$$

Therefore for t-bit fixed point arithmetic we can expect

$$|e(\mu_i)| \le \frac{\sqrt{18\,s\,n}}{\sqrt{6}} 2^{-t} = \sqrt{3\,s\,n}\; 2^{-t} \tag{5.4.3.2}$$

In comparison the Wilkinson analysis gives a bound of

$$|e(\mu_i)| \le \sqrt{2}\; s\,n^2\, 2^{-t} \tag{5.4.3.3}$$

We see that the new bound is $O(n^{3/2})$ lower than the old bound. This is precisely what we want since the old bound's dependence on n was too strong.

5.4.4 Statistical Bound for t-bit Fixed Point CORDIC Arithmetic

For t-bit fixed point CORDIC arithmetic the multiply accumulate operation has a round-off error which is C (the CORDIC constant) times the sum of t independent errors each of which are uniformly distributed on the range $[-2^{-(t+1)}, 2^{-(t+1)}]$. Accordingly it can be shown that

$$s_t = C\sqrt{t}\ 2^{-t}/\sqrt{12} \tag{5.4.4.1}$$

Therefore for t-bit, fixed point CORDIC arithmetic we can expect

$$|e(\mu_i)| \le \frac{C}{\sqrt{12}}\sqrt{18\, s\, n\, t}\ 2^{-t} \le \sqrt{s\, n\, t}\ 2^{-t} \tag{5.4.4.2}$$

In comparison the old bound showed that

$$|e(\mu_i)| \le 2\, s\, n^2\, t\, 2^{-t} \tag{5.4.4.3}$$

We see that the new bound is $O(n^{3/2}\, t^{1/2})$ smaller than the old bound.

5.4.5 Statistical Bound for t-bit Floating Point Arithmetic

To analyze the floating point case we must go back to basic principles. In either a t-bit floating point add or multiply the errors are such that

$$fl(x \text{ op } y) = (x \text{ op } y)\,(1 + \varepsilon) \tag{5.4.5.1}$$

where $|\varepsilon| \le 2^{-t}$. Therefore, the multiply accumulate operation with round-off errors included is given by

$$\hat{a}_{ii}^{(k)} = [c\,\hat{a}_{ii}^{(k-1)}(1 + \varepsilon_1) - s\,\hat{a}_{ij}^{(k-1)}(1 + \varepsilon_2)]\,(1 + \varepsilon_3) \tag{5.4.5.2}$$

Expanding this equation and ignoring terms with products of errors we see that

$$\hat{a}_{ii}^{(k)} = c\,\hat{a}_{ii}^{(k-1)} - s\,\hat{a}_{ij}^{(k-1)} + c\,\hat{a}_{ii}^{(k-1)}(\varepsilon_1 + \varepsilon_3) - s\,\hat{a}_{ij}^{(k-1)}(\varepsilon_2 + \varepsilon_3) \tag{5.4.5.3}$$

The assumptions given in section 5.4.1 show that as the Jacobi algorithm progresses $|c| \to 1$, $s \to 0$, and $\hat{a}_{ij}^{(k-1)} \to 0$. Therefore the final term in the last equation is negligible. The first two terms are just the normal products which arise in the application of a rotation. The remaining term, the third one, can be identified as $d_{ii}^{(k)}$. That is

$$d_{ii}^{(k)} = c\,\hat{a}_{ii}^{(k-1)}(\varepsilon_1 + \varepsilon_3) \tag{5.4.5.4}$$

If we assume that the ε_i are uniformly distributed on the range $[-2^{-t}, 2^{-t}]$ we can show that each d_{ii} term is a zero mean random variable with standard deviation given by

$$s_t \leq 2\,c\,\hat{a}_{ii}^{(k-1)}\,2^{-t}/\sqrt{3} \tag{5.4.5.5}$$

Therefore $e(\mu_i)$ is a sum of weighted independent random variables. By the central limit theorem $e(\mu_i)$ will be a Gaussian random variable with zero mean and its variance will be the sum of the weighted variances of its components. Unfortunately we do not know the weights so we can not compute the exact variance of $e(\mu_i)$. However since $|c| \leq 1$ and $\hat{a}_{ii}^{(k-1)} \leq ||A||_F \leq 1$, we can say that $c\hat{a}_{ii}^{(k-1)}$ will be bounded by 1 and

$$s_t \leq 2 \times 2^{-t}/\sqrt{3} \tag{5.4.5.6}$$

Therefore for t-bit. floating point arithmetic we can expect

$$|e(\mu_i)| \leq 2\frac{\sqrt{18\,s\,n}}{\sqrt{3}}\,2^{-t} = \sqrt{12\,s\,n}\;2^{-t} \tag{5.4.5.7}$$

By comparison the Wilkinson analysis bounds $e(\mu_i)$ by

$$|e(\mu_i)| \leq 24\,s\,n\,2^{-t} \tag{5.4.5.8}$$

The new bound is $O(\sqrt{n})$ smaller than the old bound.

5.4.6 Performance of the Statistical Bounds

Tables 5.4.6.1 through 5.4.6.3 show the performance of the new bounds in comparison to the maximum error data provided in Tables 5.3.1 through 5.3.3 and the original theoretical bounds.

Table 5.4.6.1

Statistical Bound for the Round-off Error of the Jacobi SVD Algorithm
With t-bit, Floating Point Arithmetic

$b1 = 24\, s\, n\, 2^{-t}$ $b2 = \sqrt{12\, s\, n}\; 2^{-t}$

As a function of n (t = 23)

n	s	max $\|e(\mu_i)\|$	b1	b1/max	b2	b2/max
10	5	3.36e-06	1.43e-04	43	2.92e-06	0.87
20	6	4.81e-06	3.43e-04	71	4.52e-06	0.94
30	7	6.77e-06	6.01e-04	89	5.98e-06	0.88
40	7	7.25e-06	8.01e-04	110	6.91e-06	0.95
50	7	8.27e-06	1.00e-03	121	7.73e-06	0.93

As a function of t (n = 20)

t	s	max $\|e(\mu_i)\|$	b1	b1/max	b2	b2/max
15	6	1.27e-03	8.79e-02	69	1.16e-03	0.91
17	6	3.23e-04	2.20e-02	68	2.90e-04	0.90
19	6	8.25e-05	5.49e-03	67	7.24e-05	0.88
21	6	2.10e-05	1.37e-03	65	1.81e-05	0.86
23	6	4.78e-06	3.43e-04	72	4.52e-06	0.95
25	6	1.18e-06	8.58e-05	73	1.13e-06	0.96
27	6	2.92e-07	2.15e-05	74	2.83e-07	0.97
29	6	6.71e-08	5.36e-06	80	7.07e-08	1.05

Table 5.4.6.2

Statistical Bound for the Round-off Error of the Jacobi SVD Algorithm
With t-bit, Fixed Point CORDIC Arithmetic

$$b1 = 2\,s\,n^2\,t\,2^{-t} \qquad b2 = \sqrt{s\,n\,t}\,2^{-t}$$

As a function of n (t = 23)

n	s	max $\lvert e(\mu_i)\rvert$	b1	b1/max	b2	b2/max
10	5	3.18e-06	2.74e-03	862	4.04e-06	1.27
20	6	6.48e-06	1.32e-02	2030	6.26e-06	0.97
30	6	8.25e-06	2.96e-02	3591	7.67e-06	0.93
40	7	1.02e-05	6.14e-02	6035	9.57e-06	0.94
50	7	1.21e-05	9.60e-02	7926	1.07e-05	0.88

As a function of t (n = 20)

t	s	max $\lvert e(\mu_i)\rvert$	b1	b1/max	b2	b2/max
15	5	1.29e-03	1.83e+00	1421	1.18e-03	0.92
17	5	2.91e-04	5.19e-01	1783	3.15e-04	1.08
19	5	8.71e-05	1.45e-01	1664	8.31e-05	0.95
21	6	2.15e-05	4.81e-02	2237	2.39e-05	1.11
23	6	6.48e-06	1.32e-02	2030	6.26e-06	0.97
25	6	1.32e-06	3.58e-03	2712	1.63e-06	1.24
27	6	4.10e-07	9.66e-04	2353	4.24e-07	1.03
29	6	8.39e-08	2.59e-04	3089	1.10e-07	1.31

Table 5.4.6.3

Statistical Bound for the Round-off Error of the Jacobi SVD Algorithm
With t-bit, Fixed Point Arithmetic

$$b1 = \sqrt{2}\, s\, n^2 2^{-t} \qquad b2 = \sqrt{3\, s\, n}\; 2^{-t}$$

As a function of n (t = 23)

n	s	max $\lvert e(\mu_i)\rvert$	b1	b1/max	b2	b2/max
10	5	1.08e-06	8.43e-05	78	1.46e-06	1.35
20	6	2.03e-06	4.05e-04	199	2.26e-06	1.11
30	7	3.06e-06	1.06e-03	347	2.99e-06	0.98
40	7	2.61e-06	1.89e-03	723	3.46e-06	1.32
50	7	3.12e-06	2.95e-03	946	3.86e-06	1.24

As a function of t (n = 20)

t	s	max $\lvert e(\mu_i)\rvert$	b1	b1/max	b2	b2/max
15	4	5.10e-04	6.91e-02	135	4.73e-04	0.93
17	5	1.16e-04	2.16e-02	186	1.32e-04	1.14
19	5	2.65e-05	5.39e-03	204	3.30e-05	1.25
21	5	7.66e-06	1.35e-03	176	8.26e-06	1.08
23	5	2.01e-06	3.37e-04	168	2.06e-06	1.03
25	5	5.53e-07	8.43e-05	152	5.16e-07	0.93
27	5	1.29e-07	2.11e-05	163	1.29e-07	1.00
29	6	3.19e-08	6.32e-06	198	3.53e-08	1.11

We see that in all cases the new bounds are much tighter than the old ones. We also see that the trends in the ratios between the old bounds and the maximum errors have been eliminated by the new bounds. Some of the bound to maximum values are slightly below one for the new bounds indicating that the proportionality constants of the new error bounds are somewhat low. This is of no concern because we will be taking base 2 logarithms of the error expressions and rounding the results up to the next higher integer to compute the number of bits needed in SVD arithmetic units. This process will more than compensate for the slightly low proportionality constants. Accordingly we conclude that the round-off errors of the singular values produced by the Jacobi algorithm are bounded by:

$$|e(\mu_i)| \leq \sqrt{12\,s\,n}\;2^{-t}, \text{ for floating point arithmetic} \qquad (5.4.6.1a)$$

$$|e(\mu_i)| \leq \sqrt{3\,s\,n}\;2^{-t}, \text{ for fixed point arithmetic} \qquad (5.4.6.1b)$$

$$|e(\mu_i)| \leq \sqrt{s\,n\,t}\;2^{-t}, \text{ for fixed point CORDIC arithmetic} \qquad (5.4.6.1c)$$

We will use these bounds in Chapter 7 to determine the number of bits needed for arithmetic units used in Jacobi SVD arrays.

6.0 ERROR BOUNDS FOR THE HESTENES SVD ALGORITHM

In Chapters 4 and 5 we computed error bounds for the round-off error of the Jacobi SVD algorithm. In this chapter we will compute similar bounds for the Hestenes algorithm.

6.1 Theoretical, Wilkinson Style Bounds

In the exact Hestenes algorithm we post-multiply the A matrix by a series of plane rotation matrices to produce the updated matrix $H = U\Sigma$. Therefore, we can use the method used by Wilkinson [Wil65] to bound the round-off error in the matrix (G) that is computed with finite precision arithmetic. We will ignore the effect of computing approximate angles in our analysis since this source of error is negligible in comparison to the error incurred in applying rotations. Since the Hestenes algorithm can operate on rectangular matrices we will compute the round-off error for an m-by-n matrix.

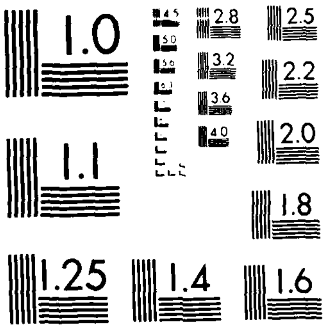
6.1.1 Wilkinson Bounds for Fixed Point and CORDIC Arithmetic

For fixed point arithmetic, when we apply a single rotation we update 2m values in the A array. If the error in each term is bounded by ε then the matrix of errors (F) caused by the application of a single rotation satisfies

$$\|F\|_F \leq \sqrt{2m}\,\varepsilon \tag{6.1.1.1}$$

In the Hestenes algorithm, a total of $n(n-1)/2$ rotations are applied in each sweep. Thus G is computed by applying a total of $N = s\,n(n-1)/2$ rotations to A where s is the number of sweeps required for convergence. Accordingly we expect the error matrix (D) between the true H and the computed version to satisfy

AD-A196 188   FINITE PRECISION ARITHMETIC IN SINGULAR VALUE   2/3
DECOMPOSITION ARCHITECTURES(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH R A DURYEA AUG 87
UNCLASSIFIED AFIT/CI/NR-88-119   F/G 12/6   NL



$$\|D\|_F = \|H - G\|_F \le s\,n\frac{(n-1)}{2}\sqrt{2m}\;\varepsilon \le \frac{\sqrt{2}}{2}s\,n^2\sqrt{m}\;\varepsilon \qquad (6.1.1.2)$$

Note that if we compute the V matrix we would expect the error in V to satisfy the same bound.

At this point we must make an assumption to proceed. In the Jacobi case, the matrix that is produced is the Σ matrix so we can say something about the *errors in the singular values. For the Hestenes algorithm we do not produce* Σ, we produce $G \approx U\Sigma$. To compute the singular values (μ_i) we use the formula

$$\mu_i = \sqrt{\mathbf{g}_i^T\mathbf{g}_i} = \|\mathbf{g}_i\|_2, \text{ for } i = 1, 2, \ldots, n \qquad (6.1.1.3)$$

However the bound given in equation 6.1.1.2 does not tell us what the size of the error is in a particular element or vector of G. We can make progress if we assume that the errors are uniformly distributed among the vectors of G.

Let's assume that $\mathbf{g}_i = \mathbf{h}_i + \mathbf{d}_i$ where $\mathbf{h}_i$ is the true vector (i.e. $\sigma_i = \|\mathbf{h}_i\|_2$) and d_i is the error vector. Therefore the error in singular value $e(\mu_i) = \sigma_i - \mu_i$ satisfies

$$|e(\mu_i)| = \big|\,\|\mathbf{h}_i\|_2 - \|\mathbf{g}_i\|_2\,\big| \le \|\mathbf{d}_i\|_2 \qquad (6.1.1.4)$$

If we assume a uniform error distribution we can say that the norm of the error vector is bounded by

$$\|\mathbf{d}_i\|_2 = \frac{1}{\sqrt{n}}\|D\|_F \le \frac{\sqrt{2}}{2}s\,n^{3/2}\sqrt{m}\;\varepsilon \qquad (6.1.1.5)$$

Combining equations 6.1.1.4 and 6.1.1.5 we see that the error in singular value $e(\mu_i)$ is bounded by

$$|e(\mu_i)| \le \frac{\sqrt{2}}{2} s\, n^{3/2} \sqrt{m}\ \varepsilon \qquad (6.1.1.6)$$

In the analysis of the Jacobi algorithm we showed that $\varepsilon = 2^{-t}$ for t-bit fixed point arithmetic. For t-bit fixed point CORDIC arithmetic $\varepsilon = [(7/6)t + 2]2^{-t}$. Substituting these values in for ε in the equation 6.1.1.6 and simplifying the result we see that

$$|e(\mu_i)| \le \frac{\sqrt{2}}{2} s\, n^{3/2} \sqrt{m}\ 2^{-t}, \text{ for t-bit fixed point arithmetic} \qquad (6.1.1.7a)$$

$$|e(\mu_i)| \le \frac{5}{6} s\, n^{3/2} \sqrt{m}\ t\, 2^{-t}, \text{ for t-bit fixed point CORDIC arithmetic} \qquad (6.1.1.7b)$$

Note that these bounds assume that no errors occur in the inner-product computation shown in equation 6.1.1.3. However, this is not necessarily the case. For example, if we use t-bit fixed point arithmetic to compute the inner products the error that occurs in the inner product computation could be as large as $\sqrt{m}\ 2^{-t/2}$. This error is potentially much larger than the error incurred in applying all of the rotations to the A matrix. To avoid this situation we will assume that the SVD array returns the entire G matrix to the host processor where the final inner product computations will be accomplished using floating point or double length fixed point arithmetic. In the case of a CORDIC array, the host (or another special purpose unit) must perform the final computations since there is no easy way to compute inner products with CORDIC AUs.

6.1.2 Wilkinson Bound for Floating Point Arithmetic

For the floating point case we can use the development given in Wilkinson for the application of a series of N rotations to matrix. Wilkinson shows [Wil65, pg 138] that the error in premultiplying an n-by-m matrix with N=n(n-1)/2 rotations is given by

$$||A_N - R_N R_{N-1} \dots R_1 A||_F \leq x\, n^{3/2} (1 + x)^{2n-4} ||A||_F \qquad (6.1.2.1)$$

where A_N is the true result and x is the error incurred in applying an approximate rotation to a pair of elements. Wilkinson bounds x by 6×2^{-t} for t-bit floating point arithmetic. As in the Jacobi analysis, this value is too small for the formulas commonly used to compute rotation parameters for the Hestenes SVD. A value of 12×2^{-t} is more appropriate. Also as we showed in the Jacobi case, the term $(1 + x)^{2n-4}$ can be ignored since in any practical case it will be ≈ 1.

The Wilkinson error bound is directly applicable to one sweep of the Hestenes algorithm (if we compute the error in A_N^T rather than A_N). If we have normalized A so that $||A||_F \leq 1$ we see that after s sweeps we have $G = A_{sN}$ and *the error in the* H *matrix will be bounded by*

$$||D||_F = ||H - AR_1 R_2 \dots R_{sN}||_F \leq 12\, s\, n^{3/2} 2^{-t} \qquad (6.1.2.2)$$

Again assuming that the errors are uniformly distributed to the vectors of D, the error in a singular value will be bounded by

$$|e(\mu_i)| \leq 12\, s\, n\, 2^{-t}, \text{ for t-bit floating point arithmetic} \qquad (6.1.2.3)$$

Note that there is no explicit dependence on the number (m) of rows in the matrix in this formula. The dependence on m is implicitly covered by the $||A||_F$ term in the Wilkinson error bound. We expect $||A||_F$ to increase with m. However, once we have normalized the matrix so that $||A||_F \leq 1$ the dependence on m is eliminated.

6.2 Statistical Bounds for the Hestenes Algorithm

Our simulation of the Jacobi algorithm showed the Wilkinson bounds to be much too loose. By using an approximate statistical analysis we were able to develop much tighter bounds which fit the experimental data well. We will use the same type of approximate statistical analysis to bound the round-off error of the Hestenes algorithm. To do so we will assume that the error characteristics of the G matrix computed by the Hestenes algorithm are the same as those of the updated A matrix produced by the Jacobi algorithm. Specifically:

a. For floating point arithmetic we will assume that the errors are relative to the value of h_{ij}, i.e. $g_{ij} = h_{ij}(1 + d_{ij})$. Based on the Jacobi analysis, we will assume that the d_{ij} are independent, normally distributed random variables with 0 mean and variance β^2 $\{N(0, \beta^2)\}$ where $3\beta = \sqrt{(6sn)}2^{-t}$. (We assume that the variance is one half that of the Jacobi algorithm since the Hestenes algorithm uses only half the number of rotations.)

b. For fixed point arithmetic we will assume that the errors are absolute errors, $g_{ij} = h_{ij} + d_{ij}$, and that the d_{ij} are independent, $N(0, \beta^2)$ where $3\beta = \sqrt{(1.5sn)}2^{-t}$.

c. For fixed point CORDIC arithmetic we will assume that the errors are absolute errors, $g_{ij} = h_{ij} + d_{ij}$, and that the d_{ij} are independent, $N(0, \beta^2)$ and that $3\beta = \sqrt{(0.5snt)}2^{-t}$.

6.2.1 Statistical Bounds for Floating Point Arithmetic

We want to compute

$$\sigma_j = \sqrt{h_j^T h_j} \quad \text{for } j = i, 2, \ldots, n \qquad (6.2.1.1)$$

Instead we compute

$$\mu_j = \sqrt{\mathbf{g}_j^T \mathbf{g}_j} \quad \text{for } j = 1, 2, \ldots, n \tag{6.2.1.2}$$

We want to analyze the error $e(\mu_j) = \sigma_j - \mu_j$. However, the square roots make it very difficult to analyze μ_j and σ_j directly. Therefore we will look at the square of these quantities.

From the definition of g_{ij} for floating point arithmetic, we see that

$$\mathbf{g}_j = (I_m + D)\, \mathbf{h}_j \tag{6.2.1.3}$$

where $D = \mathrm{diag}(d_{1j}, d_{2j}, \ldots, d_{mj})$. Accordingly

$$\mu_j^2 = \mathbf{g}_j^T \mathbf{g}_j = \mathbf{h}_j^T (I_m + D)(I_m + D)\mathbf{h}_j = \mathbf{h}_j^T \mathbf{h}_j + 2\mathbf{h}_j^T D \mathbf{h}_j + \mathbf{h}_j^T D^2 \mathbf{h}_j$$

$$= \sigma_j^2 + 2\, \mathbf{h}_j^T D\, \mathbf{h}_j + \mathbf{h}_j^T D^2 \mathbf{h}_j \tag{6.2.1.4}$$

or, expanding the vector-matrix-vector products into sums,

$$\mu_j^2 = \sigma_j^2 + 2 \sum_{i=1}^{m} h_{ij}^2 d_{ij} + \sum_{i=1}^{m} h_{ij}^2 d_{ij}^2 \tag{6.2.1.5}$$

We see that the error in μ_j^2 is made up of two components.

The first is $2\mathbf{h}_j^T \mathbf{D} \mathbf{h}_j$. Note that since we have assumed that the d_{ij} are all $N(0, \beta^2)$, then this term will be a weighted sum of normally distributed random variables. It will have a mean of 0 and a variance of $4\beta^2$ times the sum of the squares of the weights. That is

$$\mathrm{var}[2\mathbf{h}_j^T D \mathbf{h}_j] = 4\beta^2 \sum_{i=1}^{m} h_{ij}^4 \leq 4\beta^2 \sum_{i=1}^{m} h_{ij}^2 = 4\,\beta^2 \sigma_j^2 \tag{6.2.1.6}$$

So $2\mathbf{h}_j^T \mathbf{D} \mathbf{h}_j$ is approximately $N(0, 4\beta^2 \sigma_j^2)$. Its standard deviation is $\approx 2\beta\sigma_j$ Therefore 99% of the possible values of $2\mathbf{h}_j^T \mathbf{D} \mathbf{h}_j$ will lie within 3 standard deviations of its mean or, equivalently, in the range $[-6\beta\sigma_j, 6\beta\sigma_j]$.

The second component of the error in μ_j^2 is $\mathbf{h}_j^T\mathbf{D}^2\mathbf{h}_j$. Since each of the d_{ij} is $N(0, \beta^2)$, d_{ij}^2 is a chi-squared random variable with mean β^2 and variance $2\beta^4$. Therefore $\mathbf{h}_j^T\mathbf{D}^2\mathbf{h}_j$ is a weighted sum of chi-squared random variables. It is easy to show that its mean is $\sigma_j^2\beta^2$ and its variance is $\leq 2\sigma_j^2\beta^4$. Therefore, the magnitude of this component will be $O(\sigma_j^2\beta^2)$. As a result we see that this term will be negligible in comparison to the $2\mathbf{h}_j^T\mathbf{D}\mathbf{h}_j$ term which is $O(\beta\sigma_j)$.

So ignoring the $\mathbf{h}_j^T\mathbf{D}^2\mathbf{h}_j$ term we see that

$$\mu_j^2 \leq \sigma_j^2 + 6\beta\sigma_j \tag{6.2.1.7}$$

Since μ_j^2 satisfies this equation we can add any positive number to the right hand side and the equation will still be satisfied. So let's add the number $9\beta^2$ to obtain

$$\mu_j^2 \leq \sigma_j^2 + 6\beta\sigma_j + 9\beta^2 = (\sigma_j + 3\beta)^2 \tag{6.2.1.8}$$

Therefore

$$\mu_j \leq \sigma_j + 3\beta \tag{6.2.1.9}$$

and

$$|e(\mu_j)| = |\mu_j - \sigma_j| \leq 3\beta \tag{6.2.1.10}$$

We see that $e(\mu_j)$ will be $O(\beta)$. By assumption $3\beta = \sqrt{}(6sn)2^{-t}$ so we expect

$$|e(\mu_j)| \leq \sqrt{6sn}\, 2^{-t}, \text{ for floating point arithmetic} \tag{6.2.1.11}$$

6.2.2 Statistical Bounαs for Fixed Point and CORDIC Arithmetic

From the definition of g_{ij} for fixed point arithmetic, we see that

$$\mathbf{g}_j = \mathbf{h}_j + \mathbf{d}_j \tag{6.2.2.1}$$

where $\mathbf{d}_j = (d_{1j}, d_{2j}, ..., d_{mj})^T$. Accordingly

$$\mu_j^2 = \mathbf{g}_j^T\mathbf{g}_j = (\mathbf{h}_j^T + \mathbf{d}_j^T)(\mathbf{h}_j + \mathbf{d}_j) = \mathbf{h}_j^T\mathbf{h}_j + 2\mathbf{h}_j^T\mathbf{d}_j + \mathbf{d}_j^T\mathbf{d}_j$$

$$= \sigma_j^2 + 2\,\mathbf{h}_j^T\mathbf{d}_j + \mathbf{d}_j^T\mathbf{d}_j \tag{6.2.2.2}$$

Expanding the vector products into sums

$$\mu_j^2 = \sigma_j^2 + 2\sum_{i=i}^{m} h_{ij}d_{ij} + \sum_{i=1}^{m} d_{ij}^2 \tag{6.2.2.3}$$

we see that the error in μ_j^2 is made up of two components. The first, $2\mathbf{h}_j^T\mathbf{d}_j$, is a cross term between the true $\mathbf{h}_j$ vector and its associated error vector $\mathbf{d}_j$. Note that since we have assumed that the d_{ij} are all $N(0, \beta^2)$, then this cross term will be a weighted sum of normally distributed random variables. It will have a mean of 0 and a variance of $4\beta^2$ times the sum of the squares of the weights. In this case the weights are the elements of $\mathbf{h}_j$ and the sum of their squares is σ_j^2. That is, $2\mathbf{h}_j^T\mathbf{d}_j$ is $N(0, 4\beta^2\sigma_j^2)$ and its standard deviation is $2\beta\sigma_j$. Therefore 99% of the possible values of $2\mathbf{h}_j^T\mathbf{d}_j$ will lie in the range $[-6\beta\sigma_j, 6\beta\sigma_j]$.

The second component of the error in μ_j^2 is $\mathbf{d}_j^T\mathbf{d}_j$. Since each of the d_{ij} is $N(0, \beta^2)$, each element of d_{ij}^2 is a chi-squared random variable with mean β^2 and variance $2\beta^4$. Therefore $\mathbf{d}_j^T\mathbf{d}_j$ is a chi-squared random variable with m degrees of freedom. It will have mean $m\beta^2$ and variance $2m\beta^4$. Approximately 99% of the possible values of $\mathbf{d}_j^T\mathbf{d}_j$ will lie in the range $[0, 2m\beta^2]$.

From the characteristics of the two error terms we can see that unless the singular value being computed is very small, the cross term will dominate. For very small singular values the $\mathbf{d}_j^T\mathbf{d}_j$ can dominate but the magnitude of both error terms will be very small. If we ignore the $\mathbf{d}_j^T\mathbf{d}_j$ term we see that

$$\mu_j^2 \le \sigma_j^2 + 6\beta\sigma_j \qquad (6.2.2.4)$$

This is exactly the same relationship we found for floating point arithmetic in section 6.2.1. Therefore we can conclude that

$$|e(\mu_j)| = |\mu_j - \sigma_j| \le 3\beta \qquad (6.2.2.5)$$

Using the assumptions that $3\beta = \sqrt{(1.5sn)}2^{-t}$ for fixed point arithmetic and $\sqrt{(0.5snt)}2^{-t}$ for fixed point arithmetic CORDIC arithmetic we can conclude that

$$|e(\mu_j)| \le \sqrt{1.5\,s\,n}\;2^{-t}, \text{ for fixed point arithmetic} \qquad (6.2.2.6)$$

$$|e(\mu_j)| \le \sqrt{0.5\,s\,n\,t}\;2^{-t}, \text{ for fixed point CORDIC arithmetic} \qquad (6.2.2.7)$$

6.3 Summary of Hestenes Bounds

The bounds computed in section 6.2.1 and 6.2.2 are summarized below.

<u>Theoretical Bounds</u>

$$|e(\mu_i)| \le 12\,s\,n\,2^{-t}, \text{ for t-bit floating point arithmetic} \qquad (6.3.1a)$$

$$|e(\mu_i)| \le \frac{\sqrt{2}}{2}\,s\,n^{3/2}\sqrt{m}\;2^{-t}, \text{ for t-bit fixed point arithmetic} \qquad (6.3.1b)$$

$$|e(\mu_i)| \le \frac{5}{6}\,s\,n^{3/2}\sqrt{m}\;t\,2^{-t}, \text{ for t-bit fixed point CORDIC arithmetic} \qquad (6.3.1c)$$

Statistical Bounds

$$|e(\mu_i)| \leq \sqrt{6\,s\,n}\;2^{-t}, \text{ for t-bit floating point arithmetic} \qquad (6.3.2a)$$

$$|e(\mu_i)| \leq \sqrt{1.5\,s\,n}\;2^{-t}, \text{ for t-bit fixed point arithmetic} \qquad (6.3.2b)$$

$$|e(\mu_i)| \leq \sqrt{0.5\,s\,n\,t}\;2^{-t}, \text{ for t-bit fixed point CORDIC arithmetic} \qquad (6.3.2c)$$

6.4 Simulation of the Hestenes Algorithm

We performed a series of computer simulations of the Hestenes algorithm, similar to the simulations of the Jacobi algorithm, using floating point, fixed point and CORDIC arithmetic. In this section we described the simulation programs and their results.

6.4.1 Simulation Programs

The specific version of the Hestenes algorithm used in the simulations is shown in Figure 6.4.1.1. The algorithm operates on an m-by-n matrix, A. During each sweep of the algorithm all n(n-1)/2 column pairs are orthogonalized. Sweeps are performed until the measure "off" falls below a preset threshold, "delta", or until a "maxsweeps" limit is reached. Once the A matrix has been orthogonalized the singular values are computed by finding A's column norms.

The Hestenes algorithm was implemented using the same basic functions that were used in the Jacobi simulation. The only additions were functions to compute column inner products. A t-bit, floating point inner product routine was used in the floating point simulation and a t-bit, fixed point inner product routine was used in both the fixed point and CORDIC simulation. As shown at the end of section 6.1.1, the final inner products computations needed to compute the

```
$\text{off} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left[ \sum_{k=1}^{m} a_{ki} a_{kj} \right]^2$
maxsweeps = 10
sweeps = 0
delta = 1.0e-24 * off
while (off > delta  and  sweeps ≤ maxsweeps) do
        for  i = 1, 2, ..., n - 1
                for  j = i+1, ..., n
                        $r_i = \mathbf{a}_i^T \mathbf{a}_i; \quad r_j = \mathbf{a}_j^T \mathbf{a}_j; \quad g = \mathbf{a}_i^T \mathbf{a}_j$
                        $w = \frac{r_j - r_i}{2g}$
                        $t = \frac{\text{sign}(w)}{|w| + \sqrt{1 + w^2}}$
                        $c = \frac{1}{\sqrt{1 + t^2}}$
                        $s = ct$
                        for  k = 1, 2, ..., m
                                $\begin{bmatrix} a_{ki} & a_{kj} \end{bmatrix} = \begin{bmatrix} a_{ki} & a_{kj} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$
        end  { for i and j}
        $\text{off} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left[ \sum_{k=1}^{m} a_{ki} a_{kj} \right]^2$
        sweeps = sweeps + 1
end  { while }
for  i = 1, 2, ..., n
        $\sigma_i = \sqrt{\mathbf{a}_i^T \mathbf{a}_i}$
```

Figure 6.4.1.1: Hestenes SVD algorithm used in the simulation

singular values should not be performed with t-bit fixed point arithmetic. In our simulations the singular values were computed with double precision inner products.

For the Hestenes algorithm we computed the error in the singular values as a function of n, m and t. We varied n from 10 to 50 while holding m constant at 50 and t constant at 23. The value of m was allowed to range from 20 to 100 while n was held at 20 and t at 23. Finally t was varied from 15 to 29 while m and n were held at 50 and 20, respectively. For each set of variables a sufficient number of matrices were processed to generate a sample size of approximately 200 singular values.

6.4.2 Simulation Results

The results of the Hestenes simulations are shown in Tables 6.4.2.1 through 6.4.2.3. Each of the tables has three separate sections, one each to display the error data as a function of m, n and t. Each section shows the maximum error in the singular values, the theoretical and statistical bounds and the bound to maximum (b/m) ratios.

As was the case for the Jacobi algorithm, all three tables show that the Wilkinson style theoretical bounds are much too high. For all three types of arithmetic these bounds are too strongly related to n and m. For the CORDIC arithmetic, the theoretical bound is also too strongly related to t. The statistical bounds are much tighter and the ratios between these bounds and the maximum error show no significant trends. Therefore the functional relationships of the statistical bounds to m, n and t appears to be correct.

The proportionality constants of the statistical bounds for the floating point and fixed point cases were somewhat off. Table 6.4.2.1 shows that the statistical

Table 6.4.2.1

Round-off Error of the Hestenes SVD Algorithm
With t-bit, Floating Point Arithmetic

$b1 = 12sn2^{-t}$ $b2 = \sqrt{6sn}\,2^{-t}$ $b3 = \sqrt{s\,n}\;2^{-t}$

As a function of n (t = 23, m = 50)

n	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
10	5	6.46e-07	7.15e-05	111	2.06e-06	3.20	8.43e-07	1.31
20	6	1.21e-06	1.72e-04	142	3.20e-06	2.64	1.31e-06	1.08
30	6	1.35e-06	2.57e-04	190	3.92e-06	2.90	1.60e-06	1.18
40	7	1.81e-06	4.01e-04	222	4.89e-06	2.70	1.99e-06	1.10
50	7	1.99e-06	5.01e-04	251	5.46e-06	2.74	2.23e-06	1.12

As a function of m (t = 23, n = 20)

m	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
20	6	1.20e-06	1.72e-04	143	3.20e-06	2.66	1.31e-06	1.08
40	6	1.23e-06	1.72e-04	140	3.20e-06	2.59	1.31e-06	1.06
60	6	1.21e-06	1.72e-04	142	3.20e-06	2.65	1.31e-06	1.08
80	6	1.20e-06	1.72e-04	143	3.20e-06	2.67	1.31e-06	1.09
100	6	8.48e-07	1.72e-04	202	3.20e-06	3.77	1.31e-06	1.54

As a function of t (m = 50, n = 20)

t	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
15	5	2.57e-04	3.66e-02	143	7.48e-04	2.91	3.05e-04	1.19
17	5	6.40e-05	9.16e-03	143	1.87e-04	2.92	7.63e-05	1.19
19	5	1.64e-05	2.29e-03	140	4.67e-05	2.85	1.91e-05	1.16
21	6	4.29e-06	6.87e-04	160	1.28e-05	2.99	5.22e-06	1.22
23	6	9.92e-07	1.72e-04	173	3.20e-06	3.22	1.31e-06	1.32
25	6	2.17e-07	4.29e-05	197	8.00e-07	3.68	3.26e-07	1.50
27	6	5.89e-08	1.07e-05	182	2.00e-07	3.39	8.16e-08	1.39
29	6	1.93e-08	2.68e-06	139	5.00e-08	2.59	2.04e-08	1.06

Table 6.4.2.2

Round-off Error of the Hestenes SVD Algorithm
With t-bit, Fixed Point CORDIC Arithmetic

$b1 = \frac{5}{6} sn^{3/2}\sqrt{m}\ t\,2^{-t}$ $b2 = \sqrt{snt/2}\ 2^{-t}$

As a function of n (t = 23, m = 50)

n	s	max $\|e(\mu_i)\|$	b1	b1/m	b2	b2/m
10	4	1.65e-06	2.04e-03	1240	2.56e-06	1.55
20	5	2.58e-06	7.23e-03	2802	4.04e-06	1.57
30	5	3.70e-06	1.33e-02	3588	4.95e-06	1.34
40	6	3.45e-06	2.45e-02	7106	6.26e-06	1.81
50	6	4.44e-06	3.43e-02	7716	7.00e-06	1.58

As a function of m (t = 23, n = 20)

m	s	max $\|e(\mu_i)\|$	b1	b1/m	b2	b2/m
20	5	2.87e-06	4.57e-03	1593	4.04e-06	1.41
40	5	3.09e-06	6.46e-03	2088	4.04e-06	1.31
60	5	2.50e-06	7.91e-03	3161	4.04e-06	1.61
80	5	2.80e-06	9.14e-03	3260	4.04e-06	1.44
100	5	2.58e-06	1.02e-02	3963	4.04e-06	1.57

As a function of t (m = 50, n = 20)

t	s	max $\|e(\mu_i)\|$	b1	b1/m	b2	b2/m
15	3	5.89e-03	7.24e-01	123	6.47e-04	0.11
17	4	3.59e-04	2.73e-01	762	1.99e-04	0.55
19	4	4.61e-05	7.64e-02	1657	5.26e-05	1.14
21	5	7.82e-06	2.64e-02	3375	1.55e-05	1.98
23	5	3.09e-06	7.23e-03	2335	4.04e-06	1.31
25	5	6.81e-07	1.96e-03	2884	1.05e-06	1.55
27	6	1.65e-07	6.36e-04	3847	3.00e-07	1.81
29	6	5.78e-08	1.71e-04	2957	7.77e-08	1.35

Table 6.4.2.3

Round-off Error of the Hestenes SVD Algorithm
With t-bit, Fixed Point Arithmetic

$b1 = sn^{3/2}\sqrt{m/2}\ 2^{-t}$ $b2 = \sqrt{1.5sn}\ 2^{-t}$ $b3 = \sqrt{3sn}\ 2^{-t}$

As a function of n (t = 23, m = 50)

n	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
10	5	1.46e-06	9.42e-05	65	1.03e-06	0.71	1.46e-06	1.00
20	6	2.10e-06	3.20e-04	152	1.60e-06	0.76	2.26e-06	1.08
30	6	2.98e-06	5.88e-04	197	1.96e-06	0.66	2.77e-06	0.93
40	7	3.15e-06	1.06e-03	335	2.44e-06	0.78	3.46e-06	1.10
50	7	4.82e-06	1.48e-03	306	2.73e-06	0.57	3.86e-06	0.80

As a function of m (t = 23, n = 20)

m	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
20	5	1.86e-06	1.69e-04	91	1.46e-06	0.79	2.06e-06	1.11
40	5	1.78e-06	2.38e-04	134	1.46e-06	0.82	2.06e-06	1.16
60	5	1.90e-06	2.92e-04	154	1.46e-06	0.77	2.06e-06	1.09
80	5	2.08e-06	3.37e-04	162	1.46e-06	0.70	2.06e-06	0.99
100	5	2.17e-06	3.77e-04	174	1.46e-06	0.67	2.06e-06	0.95

As a function of t (m = 50, n = 20)

t	s	max $\lvert e(\mu_i)\rvert$	b1	b1/m	b2	b2/m	b3	b3/m
15	4	5.84e-04	5.46e-02	94	3.34e-04	0.57	4.73e-04	0.81
17	5	1.35e-04	1.71e-02	126	9.34e-05	0.69	1.32e-04	0.98
19	5	3.62e-05	4.26e-03	118	2.34e-05	0.65	3.30e-05	0.91
21	5	9.15e-06	1.07e-03	117	5.84e-06	0.64	8.26e-06	0.90
23	5	1.87e-06	2.67e-04	143	1.46e-06	0.78	2.06e-06	1.11
25	6	4.66e-07	8.00e-05	172	4.00e-07	0.86	5.65e-07	1.21
27	6	1.14e-07	2.00e-05	176	1.00e-07	0.88	1.41e-07	1.24
29	6	2.45e-08	5.00e-06	204	2.50e-08	1.02	3.53e-08	1.45

bound (b2) is consistently high by a factor of 2.5 to 3 for the floating point simulation. That is, the computed errors are about 2.5 to 3 times lower than expected. We found that by eliminating the constant factor of $\sqrt{6}$ from the statistical bound we obtained a very close match to the maximum error values over the full range of m, n and t. This gives a revised bound of $\sqrt{(sn)}2^{-t}$ for the floating point errors. Column b2 of Table 6.4.2.3 shows that the statistical bound for the fixed point Hestenes algorithm is low by a factor of 1.2 to 1.5. In this case we found that doubling the factor of 1.5 inside the square root of the bound produced a better bound. This gives a revised bound of $\sqrt{(3sn)}2^{-t}$ for the fixed point errors. Data for the revised bounds are given in the column labelled b3 in both Table 6.4.2.1 and 6.4.2.3. We do not have any specific rationale for these changes in the proportionality constants. We will see in the next chapter that when we compute the number of bits needed for SVD AUs, such small changes in the proportionality constant are insignificant since we will be taking logarithms of the proportionality constants.

7.0 NUMBER OF BITS REQUIRED FOR SVD ARITHMETIC UNITS

In this chapter we will combine the results of Chapters 3, 4, 5 and 6 to develop formulas for the number of bits (t) needed in the arithmetic units of SVD processors. Our objective is to insure that the arithmetic error does not corrupt the singular values. That is, we want to determine the value of t that will guarantee that the maximum error incurred in the computations is no greater than the quantization error already inherent in the singular values.

7.1 Summary of Errors in SVD Algorithms

In this section we summarize the formulas we have developed for the size of the quantization error and the round-off error for the singular values of a quantized data matrix.

We were able to show in Section 3.2.3 that the variance of the singular values $[s^2(\mu_i)]$ of a matrix of (b+1)-bit quantized data values satisfies

$$s^2(\mu_i) = s_b^2 \qquad (7.1.1)$$

where

$$s_b^2 = 2^{-2b}/12$$

This formula was developed under the assumption that $|a_{ij}| < 1$ for all i, j. In Chapter 4 we found it necessary to normalize the a_{ij} so that $||A||_F < 1$. This can be accomplished by dividing all a_{ij} by $\sqrt{(mn)}$ for a rectangular matrix (or by n for a square matrix). If we do so the variance of the scaled singular values will be 1/mn times the original variance. Therefore, the standard deviation $[s(\mu_i)]$ of the scaled singular values is given by

$$s(\mu_i) = \frac{2^{-b}}{\sqrt{12mn}} \text{ for an m-by-n matrix} \qquad (7.1.2a)$$

$$s(\mu_i) = \frac{2^{-b}}{\sqrt{12\ n}} \text{ for an n-by-n matrix} \qquad (7.1.2b)$$

In Chapter 5 we established the following bounds on the round off errors [$e(\mu_i)$] in the singular values for the Jacobi algorithm using t-bit arithmetic

$$|e(\mu_i)| \leq \sqrt{12\,s\,n}\ 2^{-t} \text{ for t-bit floating point arithmetic} \qquad (7.1.3a)$$

$$|e(\mu_i)| \leq \sqrt{s\,n\,t}\ 2^{-t} \text{ for t-bit fixed point CORDIC arithmetic} \qquad (7.1.3b)$$

$$|e(\mu_i)| \leq \sqrt{3\,s\,n}\ 2^{-t} \text{ for t-bit fixed point arithmetic} \qquad (7.1.3c)$$

In Chapter 6 we found the following bounds on the round off errors for the Hestenes algorithm

$$|e(\mu_i)| \leq \sqrt{s\,n}\ 2^{-t} \text{ for floating point arithmetic} \qquad (7.1.4a)$$

$$|e(\mu_i)| \leq \sqrt{s\,n\,t/2}\ 2^{-t} \text{ for t-bit fixed point CORDIC arithmetic} \qquad (7.1.4b)$$

$$|e(\mu_i)| \leq \sqrt{3\,s\,n}\ 2^{-t} \text{ for t-bit fixed point arithmetic} \qquad (7.1.4c)$$

7.2 Number of Bits Needed in SVD Systems

To establish the number of bits needed in an SVD system, we will insist only that the round-off error bound be equal to the standard deviation of the quantization noise. That is we want $|e(\mu_i)| = s(\mu_i)$. This criterion will insure that in the average case the round-off error will be several times less than the quantization error.

Equating the expressions for $|e(\mu_i)|$ and $s(\mu_i)$ given above and solving them for t we obtain the following formulas for t

<u>For the Jacobi SVD of an n-by-n matrix of (b+1)-bit quantized data</u>

$$t_{fl} = b + 1.5 \log_2(n) + 0.5 \log_2(s) + 3.6 \qquad (7.2.1a)$$

$$t_{fix} = b + 1.5 \log_2(n) + 0.5 \log_2(s) + 2.6 \qquad (7.2.1b)$$

$$t_{co} = b + 1.5 \log_2(n) + 0.5 \log_2(s) + 0.5 \log_2(t_{co}) + 1.8 \qquad (7.2.1c)$$

where the subscripts indicate the type of arithmetic.

<u>For the Hestenes SVD of an m-by-n matrix of (b+1)-bit quantized data</u>

$$t_{fl} = b + \log_2(n) + 0.5 \log_2(s\,m) + 1.8 \qquad (7.2.2a)$$

$$t_{fix} = b + \log_2(n) + 0.5 \log_2(s\,m) + 2.6 \qquad (7.2.2b)$$

$$t_{co} = b + \log_2(n) + 0.5 \log_2(s\,m) + 0.5 \log_2(t_{co}) + 1.3 \qquad (7.2.2c)$$

These equations show that the number of bits is directly related to b and logarithmically related to s, n, and m. The CORDIC equations, 7.2.1c and 7.2.2c, include an additional log(t) term on the right hand side to account for the t shifts and adds performed in each CORDIC OP. Finally each equation has a constant term which is the base two log of $\sqrt{12}$ (from the quantization error expression) times the proportionality constant of the round-off error bound. If we let $m = n$ in the Hestenes equations we see that the formulas are exactly the same as the Jacobi equations except the constant terms for the floating point and CORDIC arithmetic are lower Therefore we see that the number of bits for the Jacobi and Hestenes algorithms (processing square arrays).differ by at most 2 bits.

To obtain expressions for the total number of bits needed to compute the SVD, we must round the values of t generated by equations 7.2.1 and 7.2.2 to the next higher integer and add one bit for the sign. For the floating point AUs, we must also add bits for the exponent. We will assume that a single 8-bit byte is allocated to the floating point exponent. Accordingly, the word size (w) needed in the SVD AUs is given by

$$w_{fl} = \lceil t_{fl} \rceil + 9 \tag{7.2.3a}$$

$$w_{fix} = \lceil t_{fix} \rceil + 1 \tag{7.2.3b}$$

$$w_{co} = \lceil t_{co} \rceil + 1 \tag{7.2.3c}$$

Note that the word size we are computing here is the number of bits that we must use to store data values and transmit them between arithmetic units. The number of bits needed in some portions of the arithmetic units themselves will be higher since we have assumed that the arithmetic operations will produced correctly rounded results. We are not concerned with the additional bits needed for rounding operations since this extra hardware is standard in most of the ALU and multiplier chips which are available today. The characteristic that drives the design and size of such chips is the number of bits in the input and output words. The w values given by equation 7.2.3 are the number of bits which must be accepted and produced by an AU.

We have computed values of w_{fl}, w_{co} and w_{fix} for the Jacobi algorithm for various values of n and b. We chose b = 8, 15, or 23 to represent common signal p ocessing applications such as the 8-bit unsigned integers of image processing and the 16 or 24-bit signed values generated in seismic or hydroacoustic systems. The results are given in Table 7.2.1.

The table shows that fixed point AUs require the fewest number of bits. The number of bits needed for the fixed point AUs is 9 bits less than the number needed for floating point AUs over the full range of b and n. This difference is equal to the number of bits that have been allocated to the floating point exponent plus 1 additional bit for the higher round-off error of the floating point AUs. In effect the table shows that once the matrix has been properly normalized to prevent overflow, it is perfectly feasible to use fixed point adders and multipliers in Jacobi SVD arrays. However, this conclusion only applies to the AUs used to apply rotations. Our fixed point Jacobi programs did not simulate the impact of using fixed point arithmetic in the processors which compute rotations. Nevertheless, this is a significant result since we will see that in all SVD architectures proposed to date the AUs used to apply rotations far outnumber the AUs which compute rotations.

The table also shows that CORDIC fixed point AUs require only one or two bits more than fixed point units. For example if we are processing 16-bit signed data from an array of 20 sensors, the table shows that we need 28-bit fixed point CORDIC AUs, 27-bit fixed point AUs and 36-bit floating point AUs. If we are finding the SVD of a 1000-by-1000 array of 8-bit image samples we need 30 bits for CORDIC arithmetic, 29 bits for fixed point and 38 bits for floating point.

Table 7.2.1

Number of Bits Required for Jacobi SVD Arithmetic Units

		b = 8			b = 15			b = 23		
n	s	w_{fl}	w_{co}	w_{fix}	w_{fl}	w_{co}	w_{fix}	w_{fl}	w_{co}	w_{fix}
20	6	29	21	20	36	28	27	44	37	35
30	6	30	22	21	37	29	28	45	38	36
40	7	30	23	21	37	30	28	45	38	36
50	7	31	23	22	38	31	29	46	39	37
60	7	31	24	22	38	31	29	46	39	37
70	8	32	24	23	39	31	30	47	40	38
80	8	32	25	23	39	32	30	47	40	38
90	8	32	25	23	39	32	30	47	40	38
100	8	33	25	24	40	32	31	48	40	39
120	8	33	25	24	40	33	31	48	41	39
140	9	33	26	24	40	33	31	48	41	39
160	9	34	26	25	41	33	32	49	42	40
180	9	34	26	25	41	34	32	49	42	40
200	9	34	27	25	41	34	32	49	42	40
250	9	35	27	26	42	34	33	50	43	41
300	10	35	28	26	42	35	33	50	43	41
400	10	36	28	27	43	35	34	51	44	42
500	10	36	29	27	43	36	34	51	44	42
600	10	37	29	28	44	36	35	52	45	43
700	11	37	30	28	44	37	35	52	45	43
800	11	37	30	28	44	37	35	52	45	43
900	11	38	30	29	45	37	36	53	45	44
1000	11	38	30	29	45	38	36	53	46	44

Finally, the table shows that for typical SVD applications we need fairly large words to compute singular values accurately with the Jacobi algorithm. For example the chart shows that currently available 32-bit floating point processors will be adequate only for small arrays ($n < 100$) of 8-bit data. To process full images which typically contain more than 1000-by-1000 pixels we will need 38-bit floating point AUs. In order to process 100-by-100 arrays of 16-bit data we will need 40-bit floating point AUs or 32 bit CORDIC AUs. However we could use 32-bit fixed point AUs in the off-diagonal elements of arrays designed to handle 1000-by-1000 arrays of 8-bit, image data or 200-by-200 arrays of 16-bit seismic data.

For the Hestenes algorithm we have computed values of w_{fl}, w_{co} and w_{fix} for various values of n, m and b. The results are given in Table 7.2.2. The number of bits does not change very rapidly with m since the dependence on m is $\log_2(\sqrt{m})$. Accordingly we have computed the number of bits for only two values of m, 200 and 1000, while holding b constant at 15. Since the number of bits is directly related to b we give data for only two values of b, 8 and 15, while holding m constant at 200. The value of n ranges from 20 up to m.

Table 7.2.2 supports the exact same conclusions for the Hestenes algorithm as Table 7.2.1 does for the Jacobi algorithm. Over the full range of n, m and b fixed point arithmetic requires the fewest number of bits, CORDIC needs one or two bits more and floating point requires the most. However the difference between the floating point and fixed point words for the Hestenes algorithm are not as large as the difference for the Jacobi algorithm. The difference between the floating point and fixed point columns of Table 7.2.2 is consistently 7 bits. For the Jacobi algorithm the difference was 9 bits. The reduction results from the lower round-off error of the floating point Hestenes algorithm. Our simulations

Table 7.2.2

Number of Bits Required for Hestenes SVD Arithmetic Units

		b = 8			b = 15			b = 15		
		m = 200			m = 200			m = 1000		
n	s	W_{fl}	W_{co}	W_{fix}	W_{fl}	W_{co}	W_{fix}	W_{fl}	W_{co}	W_{fix}
20	6	29	22	21	36	30	29	37	31	30
30	7	29	23	22	36	30	29	38	32	30
40	7	30	24	23	37	31	30	38	32	31
50	7	30	24	23	37	31	30	38	32	31
60	7	30	24	23	37	31	30	39	33	31
70	8	31	25	24	38	32	31	39	33	32
80	8	31	25	24	38	32	31	39	33	32
90	8	31	25	24	38	32	31	39	33	32
100	8	31	25	24	38	32	31	39	33	32
120	8	32	25	24	39	32	31	40	34	32
140	8	32	26	25	39	33	32	40	34	33
160	8	32	26	25	39	33	32	40	34	33
180	9	32	26	25	39	33	32	40	34	33
200	9	32	26	25	39	33	32	41	35	33
250	9							41	35	34
300	9							41	35	34
400	10							42	36	34
500	10							42	36	35
600	10							42	36	35
700	10							42	36	35
800	10							43	37	35
900	10							43	37	36
1000	11							43	37	36

show it to be √12 less than the error of the Jacobi algorithm while the error for fixed point arithmetic is the same for both algorithms.

In conclusion tables 7.2.1 and 7.2.2 show that CORDIC and fixed point arithmetic can be used very effectively in SVD computations if the data matrix is properly scaled to prevent overflows. The CORDIC and fixed point AUs actually require fewer bits than floating point AUs. In effect the initial scaling of the data matrix eliminates the need for the exponent of the floating point AUs.

In the next few chapters we will look at specific SVD architectures and how we can use floating point, fixed point and CORDIC AUs in them.

8.0 ARCHITECTURES FOR VLSI, SVD PROCESSORS

Several different architectures have been proposed to compute the SVD using multiprocessor arrays. In the following sections, five different architectures will be described. Two of the them fall into the category of "linear" arrays. That is, the number of processors in the array grows as a linear function of the number of columns in the original A matrix [number of processors = O(n)]. The other three designs are "quadratic" arrays. For these architectures the number of processors grows as a product of the number of rows and columns in the matrix [number of processors = O(mn)].

In order to simplify the discussion of the architectures, it will be assumed that the A matrix is square (m=n). (In fact two of the quadratic arrays will only handle square matrices.) This assumption is not a severe limitation. The SVD of a rectangular matrix can be found by first computing its QR factorization

$$A = Q\begin{bmatrix} R \\ O \end{bmatrix} \tag{8.0.1}$$

where R is n-by-n and upper triangular. Then the SVD of R can be computed.

$$W^T RV = \Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n) \tag{8.0.2}$$

If we define U as follows

$$U = Q\begin{bmatrix} W & O \\ O & I \end{bmatrix} \tag{8.0.3}$$

we see that the SVD of A is given by $U^T AV = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ [Bre85a].

8.1 The Moreno Pipelined SVD Architecture

In his Masters thesis [Mor85], Jaime Moreno performed a thorough analysis of the computations required to perform the SVD. Based on this analysis he developed a highly pipelined architecture for the Hestenes SVD algorithm. A simplified top level flow diagram for the architecture appears in Figure 8.1.1. The processor consists of four pipelined stages; an inner product unit, a rotation angle computation unit, a rotation application unit and a column exchange unit. Each of the stages is in turn composed of pipelines of individual arithmetic units. The substructures are shown in Figure 8.1.2. The inner product unit is constructed from a binary tree with multipliers at the leaves and adders at the internal nodes. The rotation application unit is a large linear array of parallel multiplier-multiplier-adder (MMA) structures. The rotation angle computation stage is just a straight pipeline of arithmetic units. (The divisions and square roots are accomplished using iterative algorithms implemented with multipliers and adders.) The column exchange unit is a hardwired interconnection network which implements the round robin ordering scheme. Moreno shows how the number of arithmetic units in each stage can be varied so that the throughput *rate of all stages is matched.* By using one of the processors shown in Figure 8.1.1 or several of them in parallel, Moreno creates a whole family of SVD processors which can be tailored to meet specific throughput and hardware size requirements.

Moreno's design has several undesirable features for VLSI implementation. First, the design is irregular, especially the rotation computation unit. Second, the design does not scale directly as the size of the input matrix increases. For each value of m, a different number of arithmetic units is needed in the pipeline to attain optimum throughput. However, the number of arithmetic units is not simply related to m, especially for matrices with 150 or fewer rows. Finally, the design requires the rotation parameters to be broadcast to the potentially large linear array of arithmetic units in the rotation application unit.

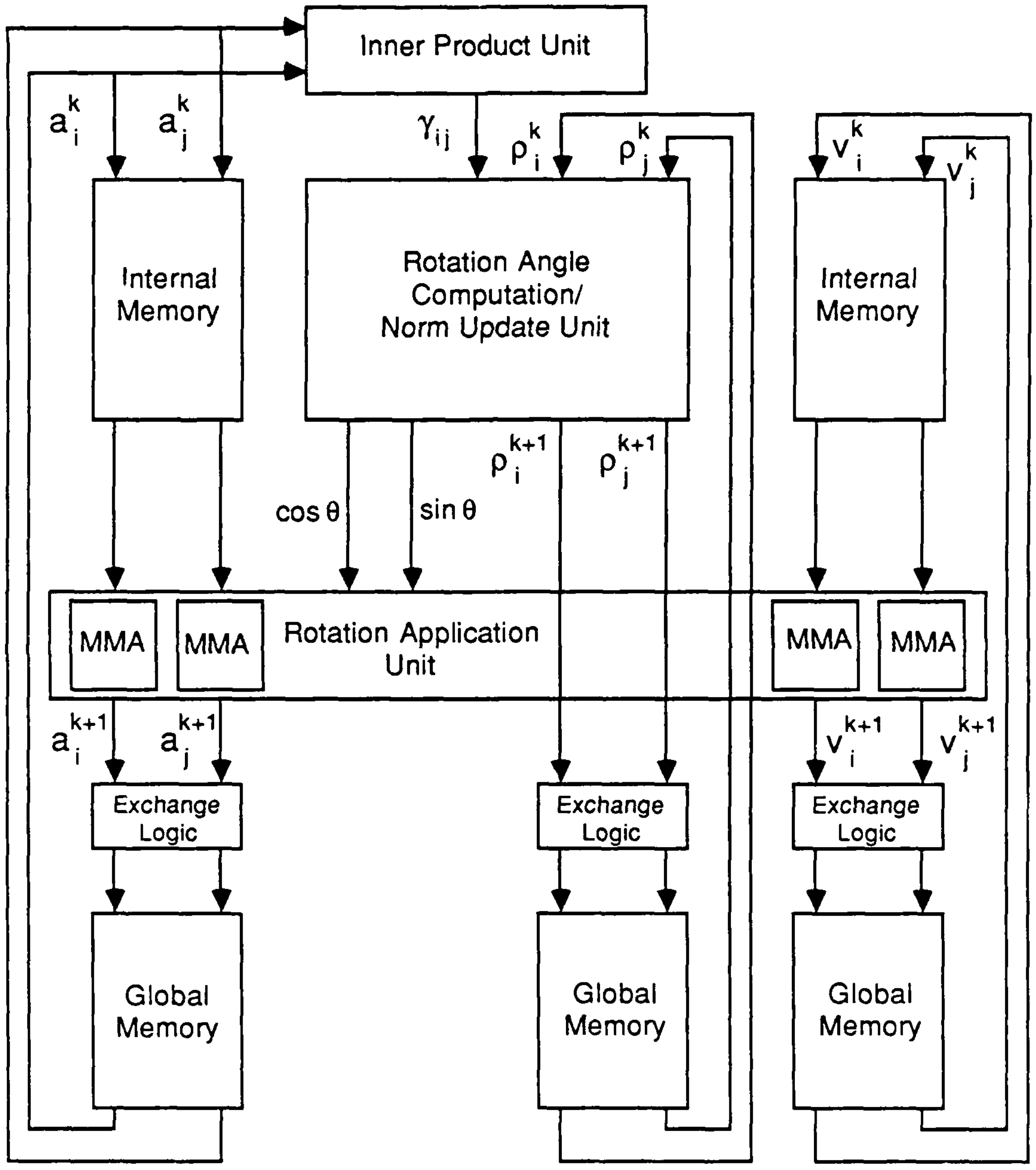
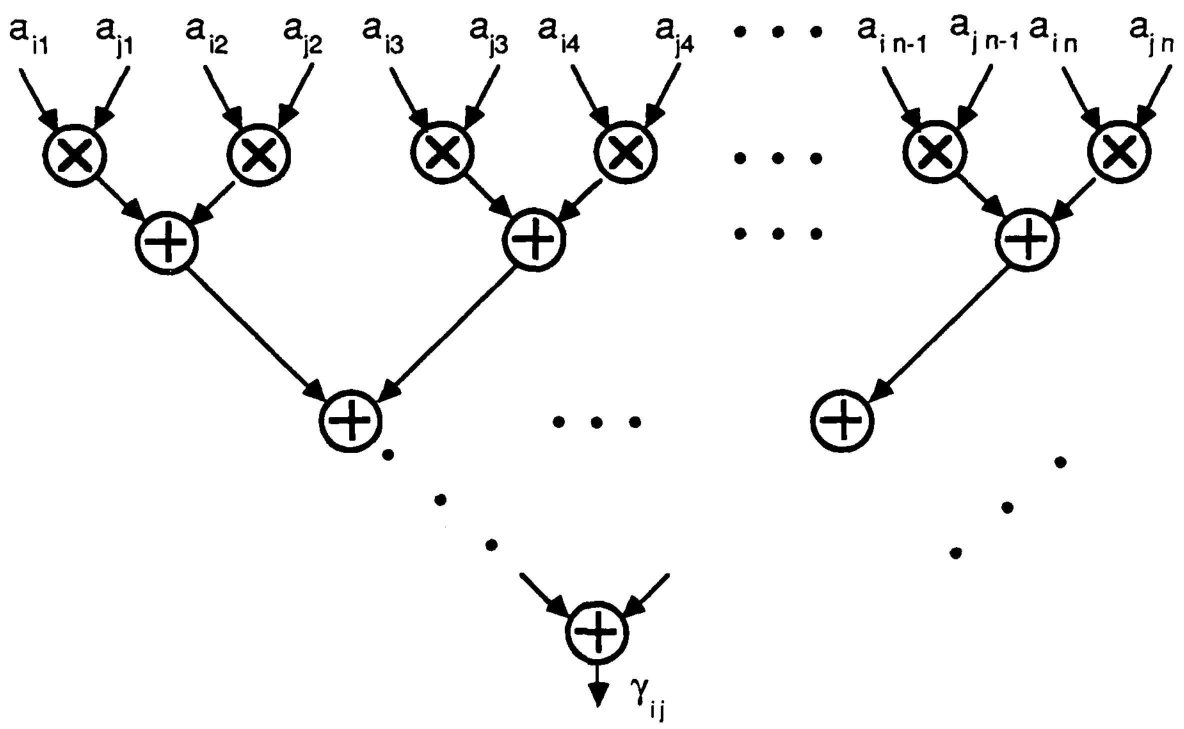


Figure 8.1.1: Moreno Pipelined SVD Architecture [Mor85]



a_{i1} a_{j1} a_{i2} a_{j2} a_{i3} a_{j3} a_{i4} a_{j4} • • • $a_{i\,n-1}$ $a_{j\,n-1}$ $a_{i\,n}$ $a_{j\,n}$
γ_{ij}

Figure 8.1.2a: Inner Product Unit [Mor 85]

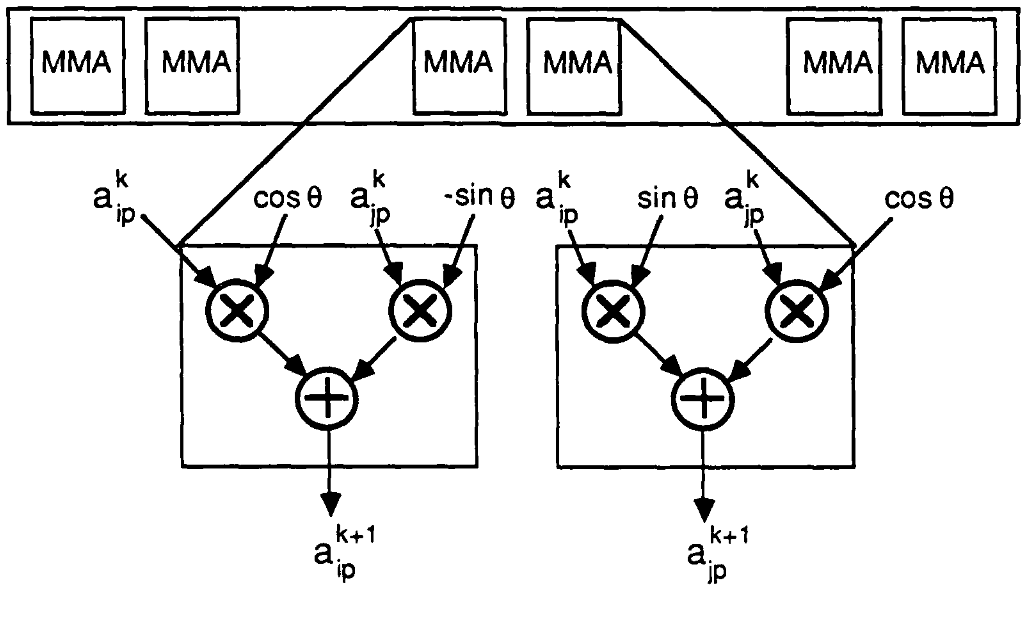


Figure 8.1.2b: Rotation application unit [Mor85]

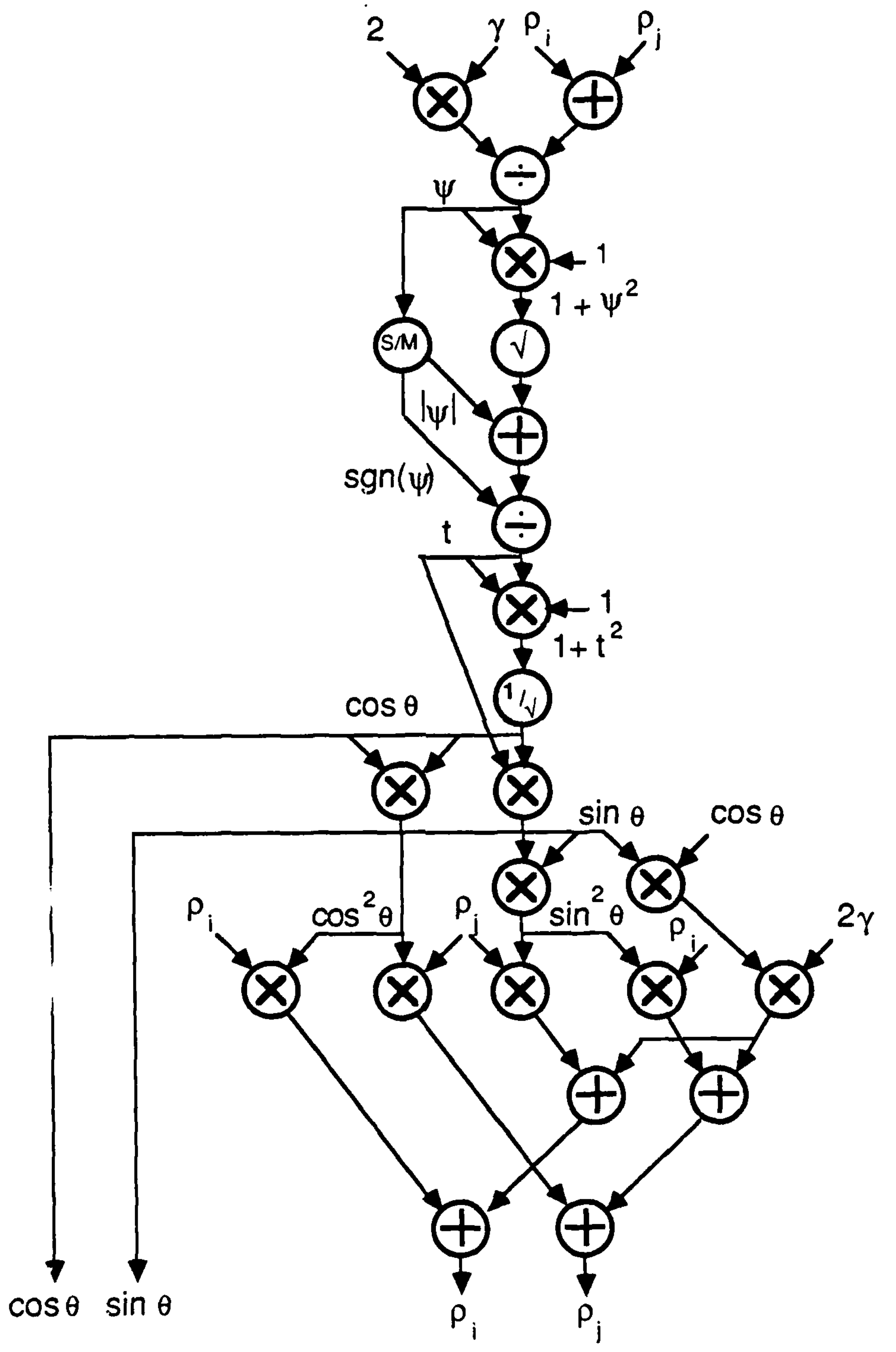


Figure 8.1.2c: Rotation angle computation / norm update unit [Mor85]

8.2 The Schimmel/Luk Linear Systolic Architecture

David Schimmel and Franklin Luk have proposed [Sch86a] an SVD architecture which on the surface is very similar to the Moreno design (see Figure 8.2.1). It too implements the Hestenes algorithm, but it uses the odd-even ordering scheme for generating rotation pairings. It also has an inner product unit, a rotation angle computation unit and a rotation application unit (matrix multiplier). However, the detailed designs of the stages are quite different (see Figure 8.2.2). For example the inner product unit is a linear systolic array of multiply accumulate cells. The rotation angle computation unit is essentially identical to that of Moreno since it is also implemented as a pipeline of arithmetic units.

The heart of the Schimmel/Luk design is the matrix multiplication unit. This unit simultaneously applies the rotations and performs the column exchange function. The basis for its design is the observation that the rotations and the column exchanges can be performed by postmultiplying the A matrix (and, if required, the V matrix) by a tridiagonal matrix of rotation parameters. Therefore the rotation application unit consists of three rows of multiply accumulate cells which perform this matrix multiplication.

The Schimmel/Luk architecture is well suited to VLSI implementation. The matrix multiplication and the inner product units are composed of regular arrays of identical multiply accumulate cells. The architecture requires only local interconnections. The design scales directly with m, the number of rows of the data matrix. It has no dependence on n. The only complicated part of the design is the rotation computation unit. However, there is only one of them and its design is the same for any size matrix.

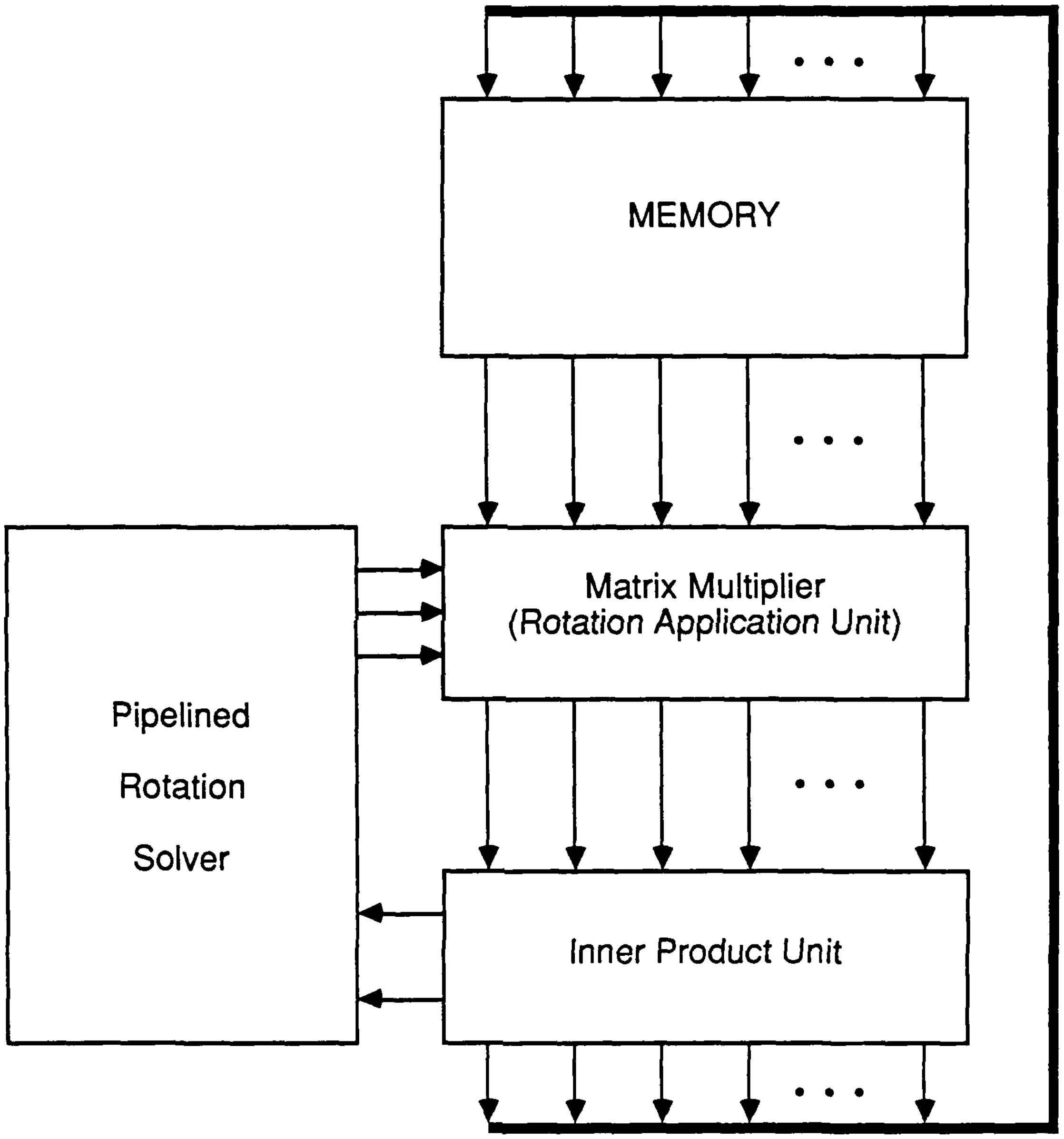


Figure 8.2.1: Schimmel / Luk SVD machine [Sch86a]

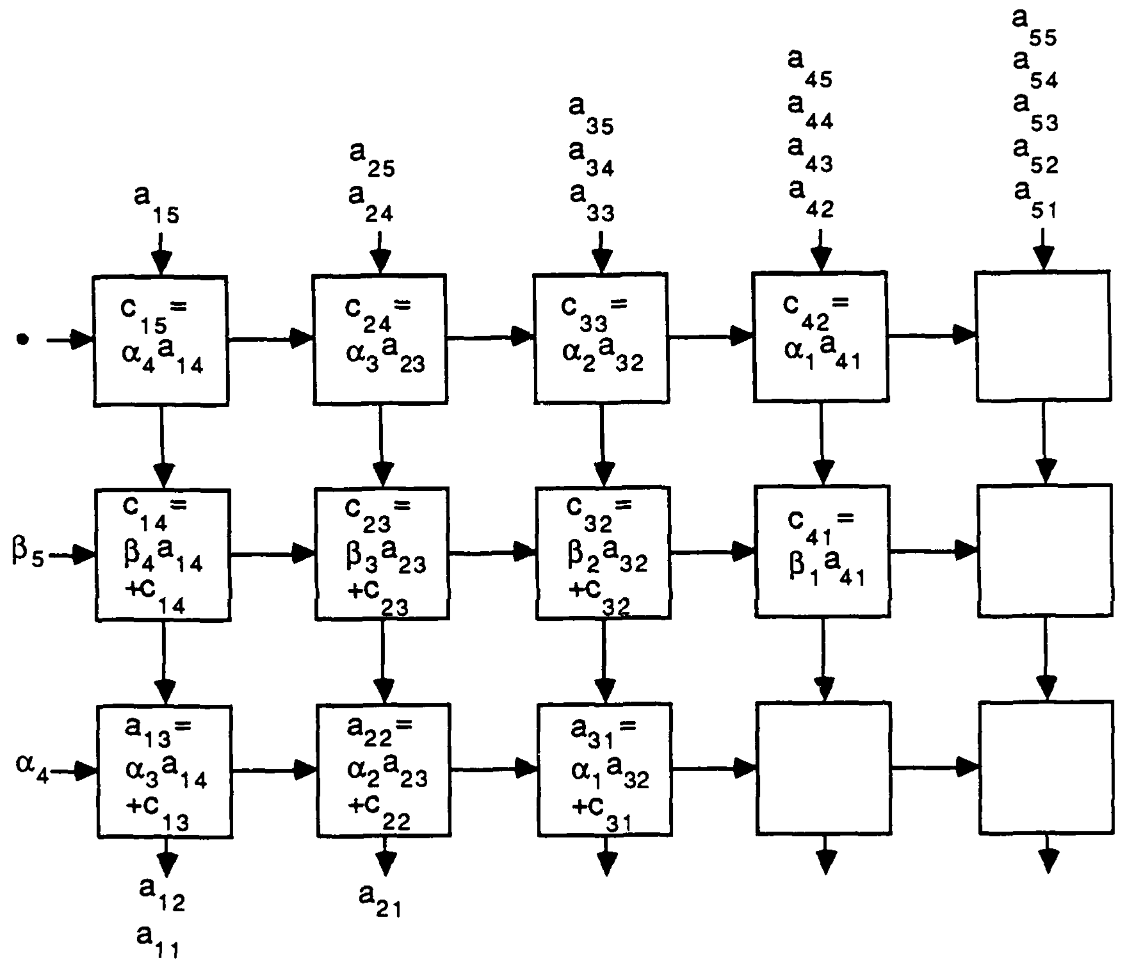
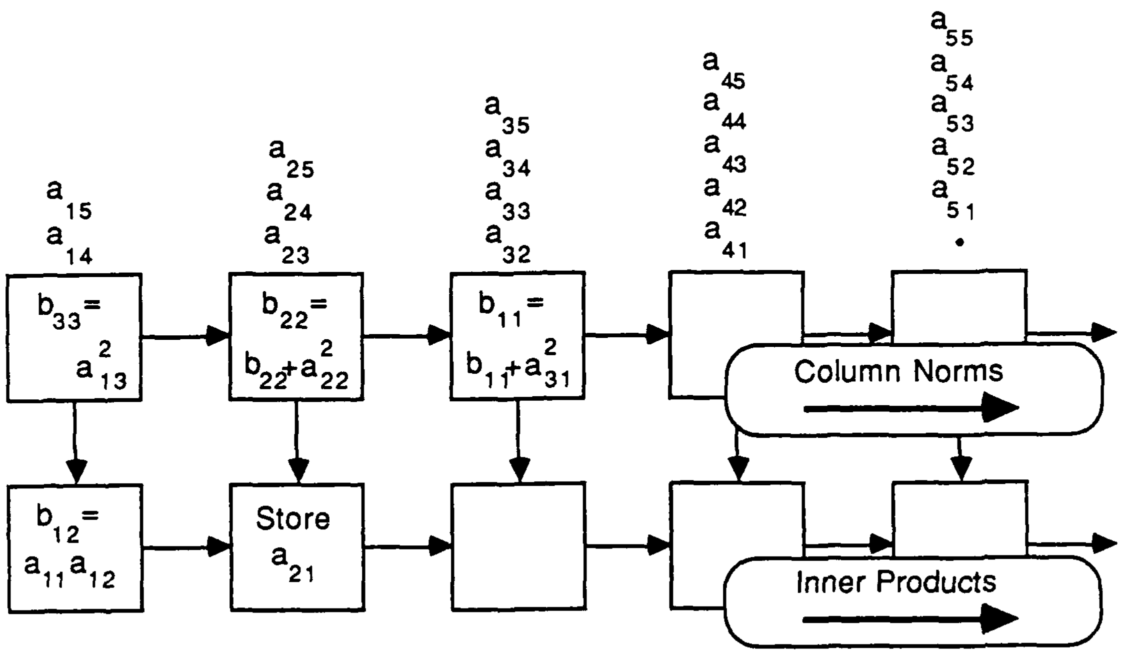

a_{15}
a_{25} a_{24}
a_{35} a_{34} a_{33}
a_{45} a_{44} a_{43} a_{42}
a_{55} a_{54} a_{53} a_{52} a_{51}
$c_{15} = \alpha_4 a_{14}$
$c_{24} = \alpha_3 a_{23}$
$c_{33} = \alpha_2 a_{32}$
$c_{42} = \alpha_1 a_{41}$
β_5
$c_{14} = \beta_4 a_{14} + c_{14}$
$c_{23} = \beta_3 a_{23} + c_{23}$
$c_{32} = \beta_2 a_{32} + c_{32}$
$c_{41} = \beta_1 a_{41}$
α_4
$a_{13} = \alpha_3 a_{14} + c_{13}$
$a_{22} = \alpha_2 a_{23} + c_{22}$
$a_{31} = \alpha_1 a_{32} + c_{31}$
a_{12} a_{11}
a_{21}

Figure 8.2.2a: Matrix multiplication unit [Sch86a]



a_{15} a_{14}
a_{25} a_{24} a_{23}
a_{35} a_{34} a_{33} a_{32}
a_{45} a_{44} a_{43} a_{42} a_{41}
a_{55} a_{54} a_{53} a_{52} a_{51}
$b_{33} = a_{13}^2$
$b_{22} = b_{22} + a_{22}^2$
$b_{11} = b_{11} + a_{31}^2$
Column Norms
$b_{12} = a_{11}a_{12}$
Store a_{21}
Inner Products

Figure 8.2.2b Inner product unit [Sch86a]

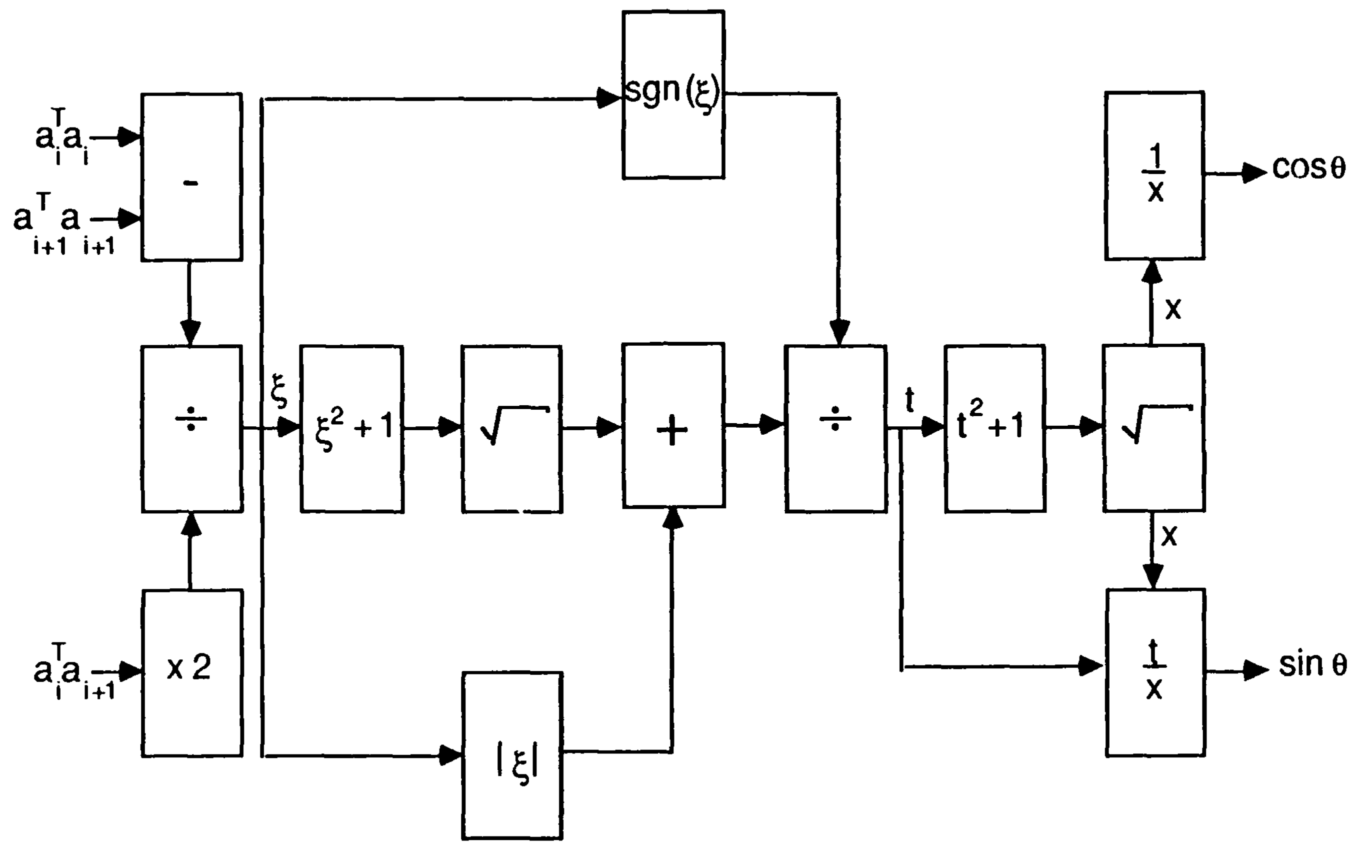


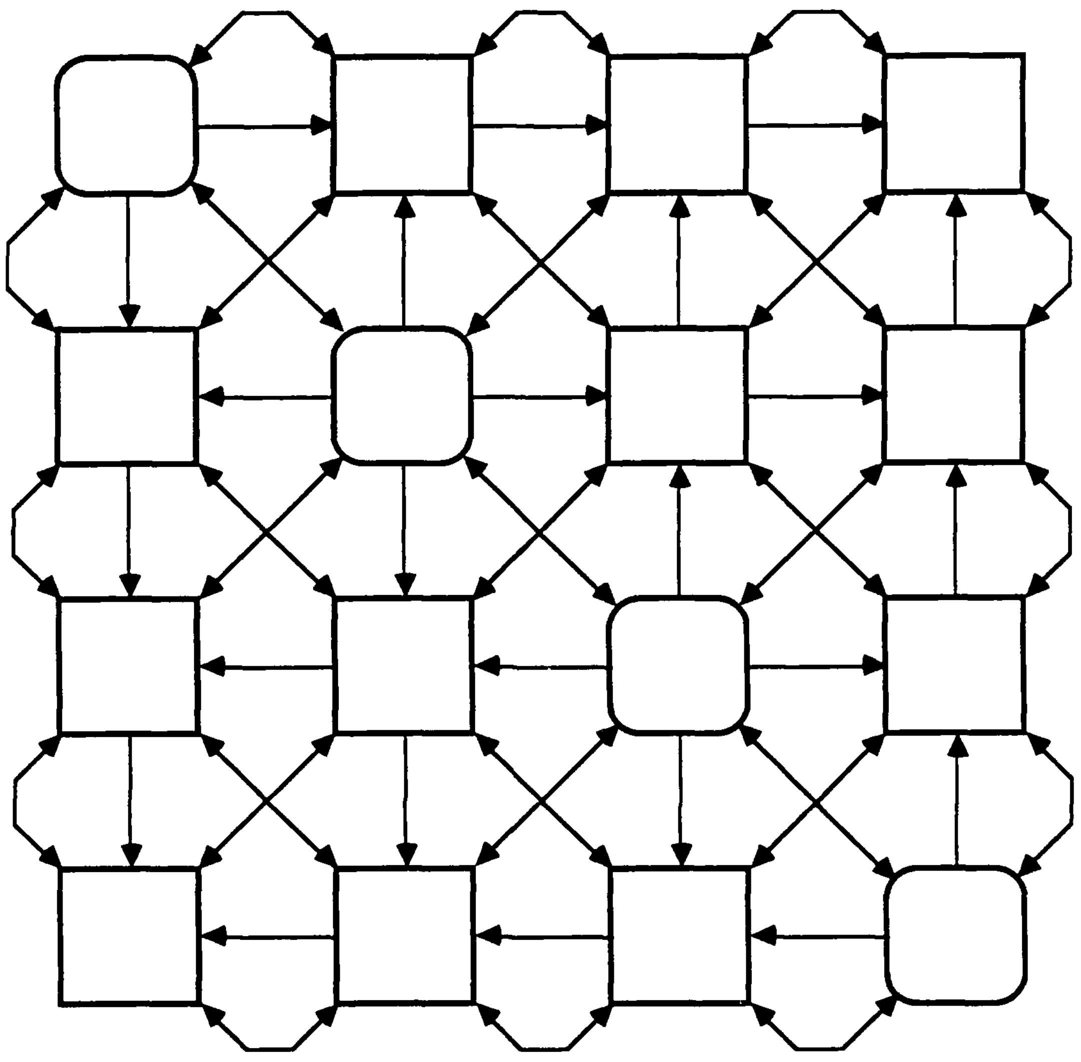
Figure 8.2.2c: Pipelined rotation solver [Sch86a]

8.3 The Brent/Luk/Van Loan (BLV) Mesh Connected Array

Richard Brent, Franklin Luk and Charles Van Loan have proposed a mesh of processors to implement the Jacobi SVD algorithm [Bre85a]. The architecture is depicted in Figure 8.3.1. It consists of an n/2-by-n/2 array of processors each of which is connected to its eight nearest neighbors. Each of the processors contains a 2-by-2 submatrix of the data matrix. The design is only defined for square data matrices.

In operation, each diagonal processor generates rotations which annihilate the off diagonal elements of its 2-by-2 submatrix (as in Figure 8.3.1). Each diagonal processor then transmits one set of rotation parameters up and down its column using the vertical mesh connections and the other set across its row using the horizontal connections. On receiving the appropriate rotation parameters, each of the off-diagonal processors applies them to its 2-by-2 submatrix (as in Figure 8.3.1). Following the application of the rotations, the processors exchange data elements in a round robin pattern in both the vertical and horizontal directions using the diagonal mesh connections. While each of these steps is described as a separate action, in actual operation the steps overlap so that every processor is active every third time step (see [Bre85a] for details). On completion, every diagonal cell holds a pair of singular values and every cell contains 2-by-2 submatrices of U and V.

This design is well suited for large scale integration due to its highly regular structure and nearest neighbor connections. The design requires two distinctly different processor types; a complex diagonal processor to compute rotations and a very simple off-diagonal processor to apply rotations. This particular architecture is probably best suited for wafer scale integration since the number of processors is quite large even for small values of n.



Diagonal elements compute c_1, s_1, c_2, and s_2

$$\begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} w & x \\ y & z \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Off-diagonal elements apply c_i, s_i, c_j, and s_j

$$\begin{bmatrix} c_i & -s_i \\ s_i & c_i \end{bmatrix} \begin{bmatrix} s & t \\ u & v \end{bmatrix} \begin{bmatrix} c_j & s_j \\ -s_j & c_j \end{bmatrix} = \begin{bmatrix} s' & t' \\ u' & v' \end{bmatrix}$$

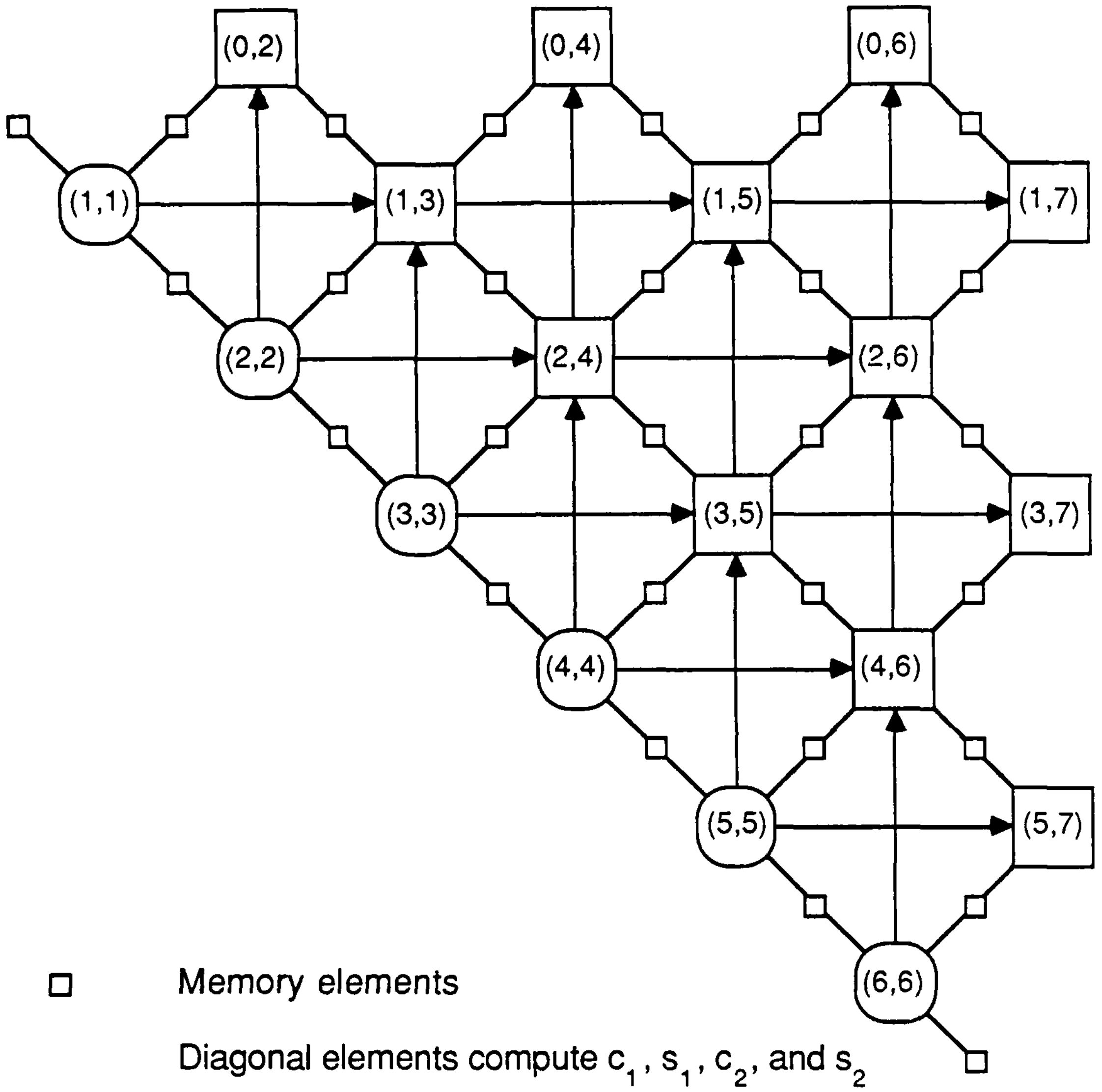
Figure 8.3.1: Brent/Luk/Van Loan mesh connected SVD array [Bre85a]

8.4 The Luk Triangular Array

Franklin Luk has independently proposed a triangular array to compute the SVD using the Jacobi algorithm [Luk86]. A top level diagram of the array appears in Figure 8.4.1. This particular architecture is unique in that it first computes the QR decomposition of the data matrix as it enters at the top of the array. The array then computes the SVD of R. (See [Luk86] for a description of the reduction of A to R.)

The architecture consists of n-1 diagonal processors which compute rotations and a triangular array of approximately $n^2/4$ off-diagonal processors which apply rotations. Located on the diagonals between processors are memory elements which store the R matrix. These memory elements are accessible by either processor connected to them. These diagonal "connections" provide for the movement of data dictated by the odd-even rotation pattern used in the array. The vertical and horizontal connections are used to transmit rotation parameters.

The array operates in the following manner to compute the SVD of R. First the odd numbered diagonal processors compute rotations to annihilate the off-diagonal elements associated with them. These rotation parameters are then transmitted up the columns and across the rows to the odd numbered off-diagonal processors. The off-diagonal processors apply the rotations to the 2-by-2 submatrices stored in the four memory elements around them. The diagonal and off-diagonal processors also permute the elements of the 2-by-2 submatrices associated with them in accordance with the odd-even rotation pattern. After the "odd" rotations have been computed and applied, the even numbered diagonal processors compute rotation parameters and transmit them to the even numbered off-diagonal processors. The process is then repeated with the odd



☐ Memory elements

Diagonal elements compute c_1, s_1, c_2, and s_2

▢ $$\begin{bmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} w & x \\ 0 & z \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Off-diagonal elements apply c_i, s_i, c_j, and s_j

□ $$\begin{bmatrix} c_i & -s_i \\ s_i & c_i \end{bmatrix} \begin{bmatrix} s & t \\ u & v \end{bmatrix} \begin{bmatrix} c_j & s_j \\ -s_j & c_j \end{bmatrix} = \begin{bmatrix} s' & t' \\ u' & v' \end{bmatrix}$$

Figure 8.4.1: Luk triangular SVD array [Luk86]

processors and so on. As with the BLV array, in actual operation these steps are overlapped. For the Luk triangular array each processor is active during every fourth time step. On termination the singular values will be stored in the memory elements along the main diagonal. If the singular vectors of A are required the triangular array must be extended to an m-by-n rectangular array so that the rotations can be accumulated (see [Luk86] for details).

The Luk triangular array has essentially the same characteristics as the Brent/Luk/Van Loan array in terms of VLSI implementation.

8.5 The Finn Triangular Array

The final SVD architecture to be discussed was developed by Alan Finn [Fin82a, Fin82b, Fin83]. The design implements an "approximate" version of the Hestenes algorithm on a triangular array of processors. Figure 8.5.1 presents the top level diagram. The array has $n^2/2$ identical processors and requires only horizontal and vertical connections. It operates in the following manner. Each sweep of the algorithm requires two full passes of the data matrix through the array. In each pass, the data elements enter from the left hand side of the array with one column entering each processor. They flow horizontally across to the diagonal processor which reflects them downward. The elements then flow off the bottom of the array and are returned to the left hand edge by the end around connections. On the first pass of the data through the array, the inner product of columns i and j is computed in processor (i,j). At the start of the second pass, processor (i,j) computes a rotation which will orthogonalize columns i and j and then applies the rotation to the elements of columns i and j as the data passes through the second time. This procedure only approximates the Hestenes algorithm because all rotations involving column k will not be complete before the

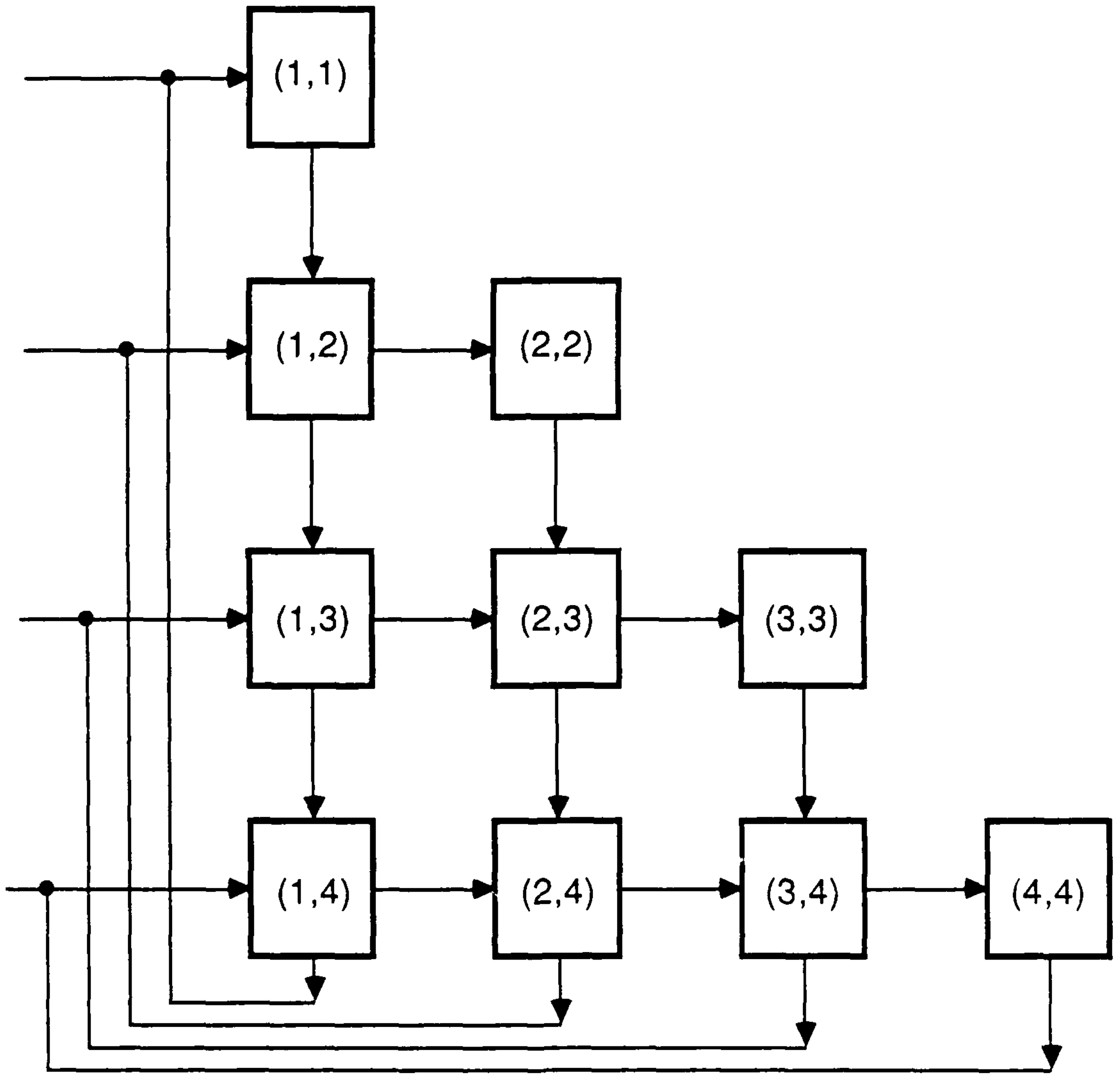


Figure 8.5.1: Finn SVD array [Fin83]

next series of inner products involving column k begin. Therefore the inner products are based on data which can be up to one sweep older than would be used in the true Hestenes algorithm [Fin82a]. Nevertheless, Finn found that his algorithm converged, although it required more sweeps than the true Hestenes method.

Of the five architectures described above, Finn's is probably the best for VLSI implementation. All of the processors in his design are identical. The design has an extremely simple interconnection pattern including very natural data paths into and out of the array. The design scales directly with increases in the number of columns in the data matrix (the number of rows has no impact on the design). The only potential problems with the design, for VLSI implementation, are the long end around connections.

9.0 RESOURCE REQUIREMENTS OF THE SVD ARCHITECTURES WITH FLOATING POINT (OR FIXED POINT) AUs

In this chapter the SVD architectures described in Chapter 8 will be analyzed to determine the number of floating point arithmetic units and the amount of time they require to compute an SVD. This analysis is based on the method used by Moreno in the development of his SVD architecture [Mor85]. The method consists of breaking the algorithm computed by each architecture into successively finer steps until you reach the level of individual multiplications and additions. At that point you can determine an appropriate number of arithmetic units to be assigned to each processor in an architecture and the computation time that will result.

9.1 Ground Rules for the Comparison

9.1.1 Definitions

a. Arithmetic Unit (AU) - A circuit element which performs either a single floating point multiply or a single floating point add. This comparison does not consider the use of pipelined arithmetic units.

b. Operation (OP) - The time required to perform a floating point multiply or a floating point add. For presently available VLSI AUs, this time is approximately 100 nanoseconds [Lei87].

c. SVD Computation time - The time required by an architecture to complete the computation of the SVD of a data matrix. The computation time is the actual "elapsed wall clock time" for the architecture to compute U, Σ and V. However, to make the times independent of VLSI hardware technology, they will be expressed in units of OPs.

d. Data matrix - In order to simplify the computations of the numbers of AUs and computation time, it will be assumed that the data matrix is square with size n-by-n.

e. Dependency Graphs - To facilitate the analysis of each architecture, a graphical technique used extensively by Moreno [Mor85] will be employed here. In this technique a hierarchical set of graphs are constructed to represent the computations which must be performed. In these graphs the nodes represent the computations and the arcs represent the precedences between them. For example, Figure 9.1.1.1 shows the top level dependency graphs for the Hestenes algorithm as implemented in Moreno's architecture.

9.1.2 Calculation of Divisions and Square Roots

Both the Hestenes and Jacobi algorithms require the computation of divisions and square roots. It will be assumed that these operations will be performed using iterative algorithms involving multiplications and additions. For his analysis, Moreno selected algorithms based on the Goldschmidt method for division [And67] and a similar method for square-root given in [Ram72]. Moreno estimates that with a sufficiently accurate initial guess (obtained by table look-up) a division can be completed in the equivalent of 9 OPs and a square-root in 12 OPs [Mor85]. Figure 9.1.2.1 shows the dependency graphs for the division and square root algorithms. The graphs show that two AUs can be used in parallel to reduce the computation time to 6 OPs for a division and 9 OPs for a square root (allowing 1/2 OP for a table look-up or a 2's complement).

Recently one manufacture (Bipolar Integrated Technology) has announced [EDN87] a floating point chip which can perform divisions and square roots in single (extended) clock cycles. The instructions are performed without the need

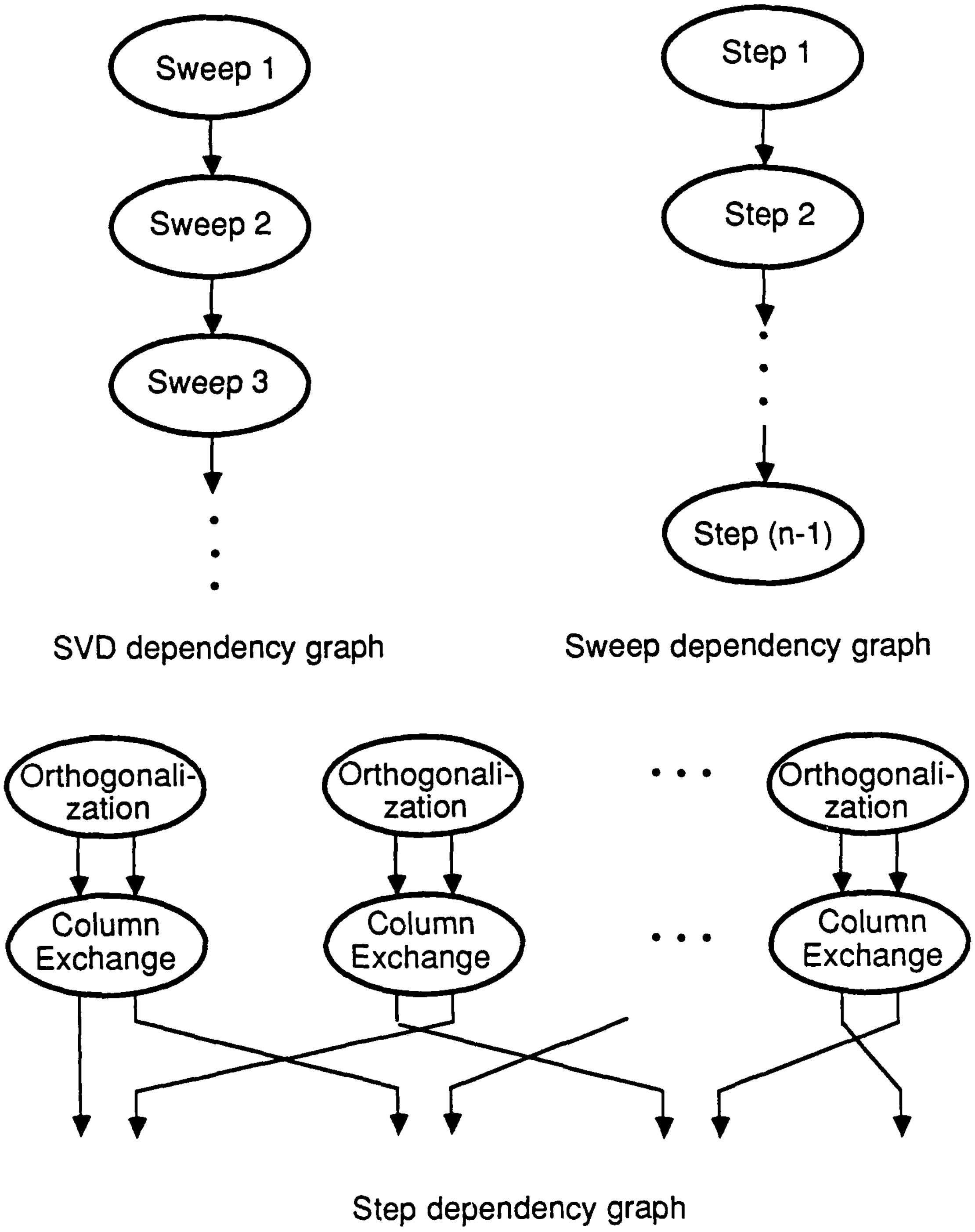


Figure 9.1.1.1: Top level dependency graphs for the Hestenes algorithm (as implemented in Moreno's architecture [Mor85])

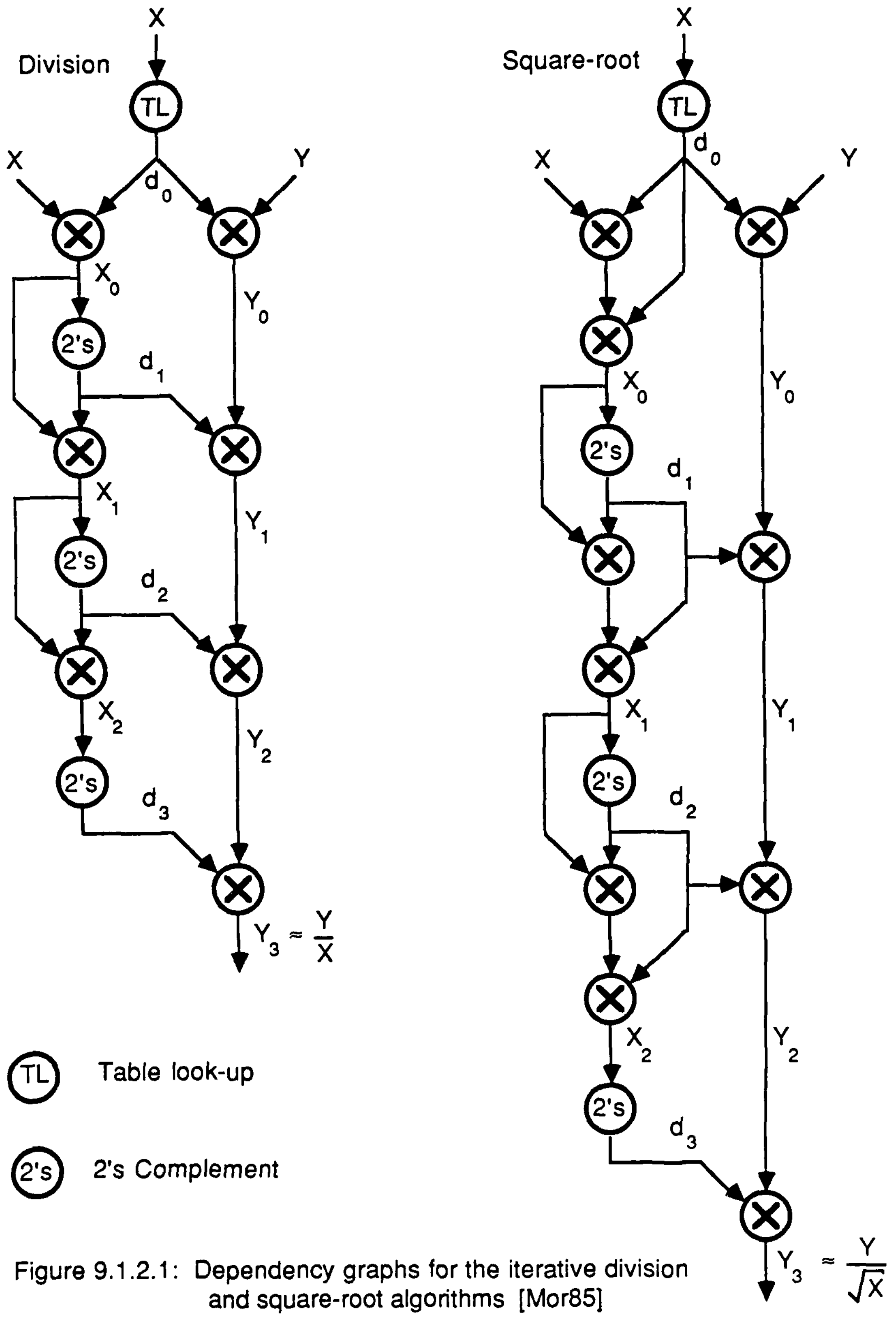


Figure 9.1.2.1: Dependency graphs for the iterative division and square-root algorithms [Mor85]

for externally provided seed values or look-up tables. The company estimates the time for a division to be 3-4 times that of a floating point multiplication. The time for a square root is at least 5-7 times the multiplication time. They give no data on the maximum time for a square root. We will not use the chip as a basis for our analysis since the technical data for it is preliminary and incomplete. However its announcement is very interesting since the chip could greatly simplify the implementation of SVD array cells which compute rotation parameters.

9.1.3 Computation Time vs Arithmetic Units

All of the architectures described in Chapter 8 allow trade-offs between the numbers of arithmetic units and computation time. To perform a meaningful comparison of the architectures it is necessary to limit these trade-offs in some manner. We have chosen to allocate arithmetic units to the architectures in a way that gives the fastest possible computation time while maintaining "reasonable" efficiency for each of the arithmetic units. In most cases the optimum number of AUs is readily apparent. However, in some cases it is possible to reduce the computation time marginally at the expense of a large increase in the number of arithmetic units. Normally in these cases, which are identified in the text, we have chosen to accept the slight increase in time.

9.1.4 Floating Point versus Fixed Point AUs

As shown in Chapter 7, if we normalize the input matrix correctly we can replace floating point AUs with fixed point multipliers and adders in the portions of the architectures which compute inner-products or apply rotations. With current technology there is little difference between the area consumption and speed of floating point and fixed point multipliers with similar word sizes. However, tables 7.2.1 and 7.2.2 show that we can use smaller fixed point words. Therefore, using fixed point AUs could reduce the area consumption of the large rotation application units which are present in every one of the architectures. The fixed point AUs could also provide a marginal speed improvement.

Our intent is to compare the number of AUs in each architecture and the number of operations required to complete an SVD computation. For this purpose it really make no difference if we use fixed point or floating point AUs since they perform identical functions. The fixed point AUs might give slightly lower computation times and area requirements but the reductions would be similar for all of the architectures. Therefore to simplify the analysis we will discuss only floating point AUs.

9.2 Number of Sweeps for Convergence

For all of the architectures discussed in Chapter 8, the SVD is computed by a series of iterations or "sweeps" through the algorithm embedded in the design. Therefore, the computation time (T) is given by:

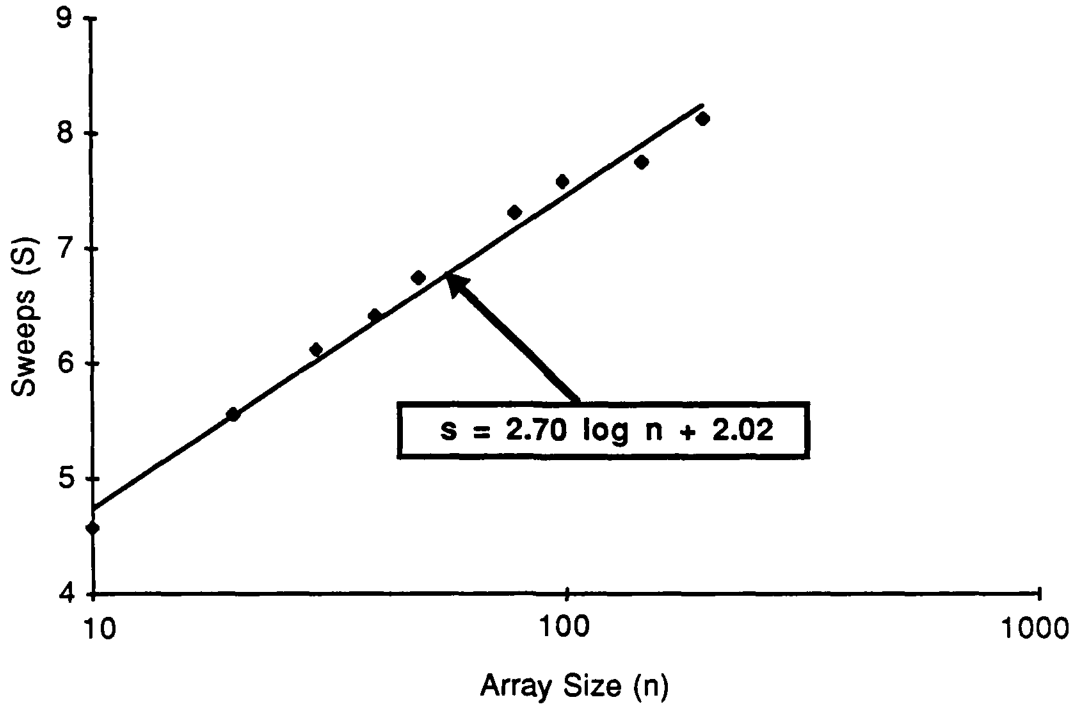
$$T = \text{\# of sweeps x time per sweep} \qquad (9.2.1)$$

The number of sweeps (s) is dictated by the performance of the algorithm. The time per sweep is dictated by the architecture and the performance of the processors within it.

Theoretically all of the SVD algorithms require an infinite number of sweeps to compute the exact U, Σ and V matrices. In practice it is not necessary to compute the "exact" values but only values accurate to within a specified working precision. Experimentally, it has been shown that both the Hestenes and Jacobi algorithms will converge to a sufficiently accurate solution in a number of sweeps that is O(log n) [Bre85a, Bre85b]. In fact rather than perform any convergence tests to terminate the algorithms, researchers have recommended just running an algorithm for a constant number (say 10) of sweeps for any size matrix up to n = 1000 [Bre85a, Bre85b]. These observations do not apply to the "approximate" Hestenes algorithm developed by Finn. He found that the number of sweeps for his algorithm was more than O(log n) [Fin83]. Therefore, to compare the computation time of the architectures, it is necessary to determine expressions for the number of sweeps for each algorithm as a function of the array size (n).

The expressions were determined by performing linear regressions on sets of experimental data available in the literature. For the Hestenes algorithm the data from Table 1 of [Bre84] was used. This data set and the best fit curve of the form $s = a \log_{10}(n) + b$ for it are shown in Figure 9.2.1. The values for a and b determined by linear regression are 2.7 and 2.0 respectively. For the Jacobi algorithm, the data set given in Table 1 (column FHSVD) of [Bre85a] was used. The data and the linear regression results for the model $s = a\log_{10}(n) + b$ are given in Figure 9.2.2. In this case the values of a and b are 3.1 and 1.5 respectively. For Finn's approximate Hestenes algorithm the data was taken from Table 4.4.2 (Method C) of [Fin83]. Several different functional models were tried. The one that appeared to fit the best was $s = bn^a$ [or equivalently $\log(s) = a \log(n) + \log(b)$]. The data and the regression results for this model are shown in Figure 9.2.3. The best values for a and b were found to be 0.64 and 1.95 respectively. That is, the number of sweeps for Finn's algorithm is $O(n^{0.64})$. This is a much more rapid rate of growth than log(n).

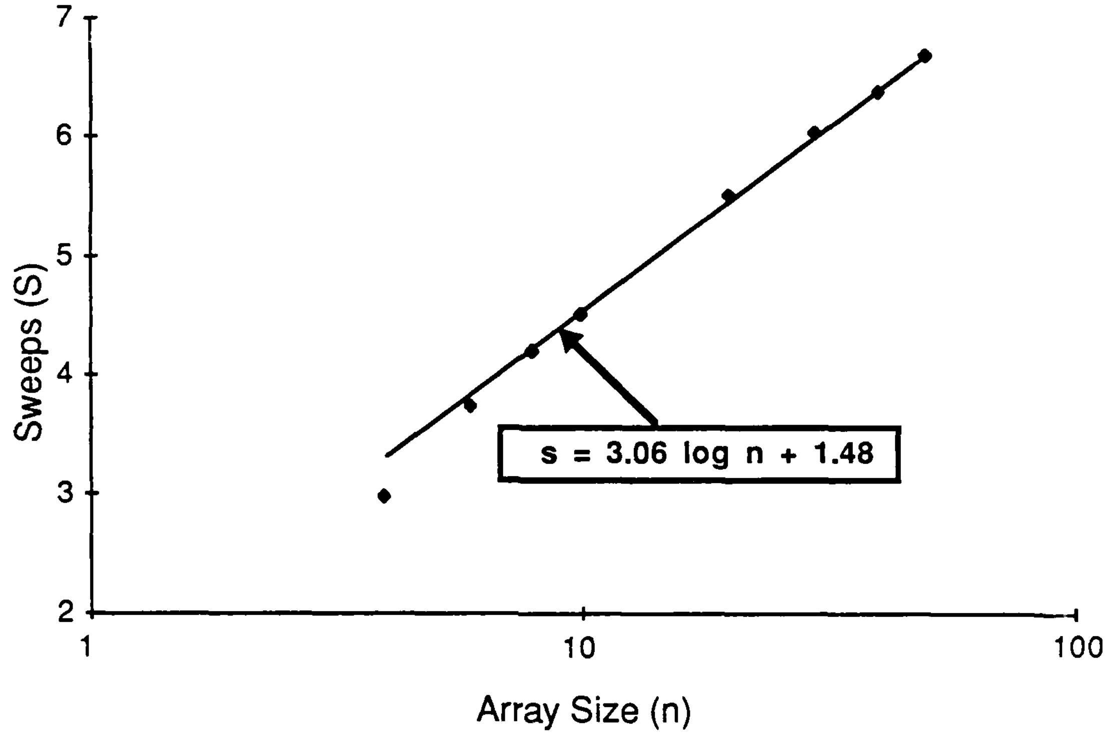
Now that we have established relationships for the number of sweeps, we will analyze each architecture to determine the time needed per sweep and the number of AUs required to achieve that time.



n	Avg. Sweeps
10	4.55
20	5.54
30	6.09
40	6.40
50	6.72
80	7.30
100	7.56
150	7.73
200	8.10

Source [Bre84]

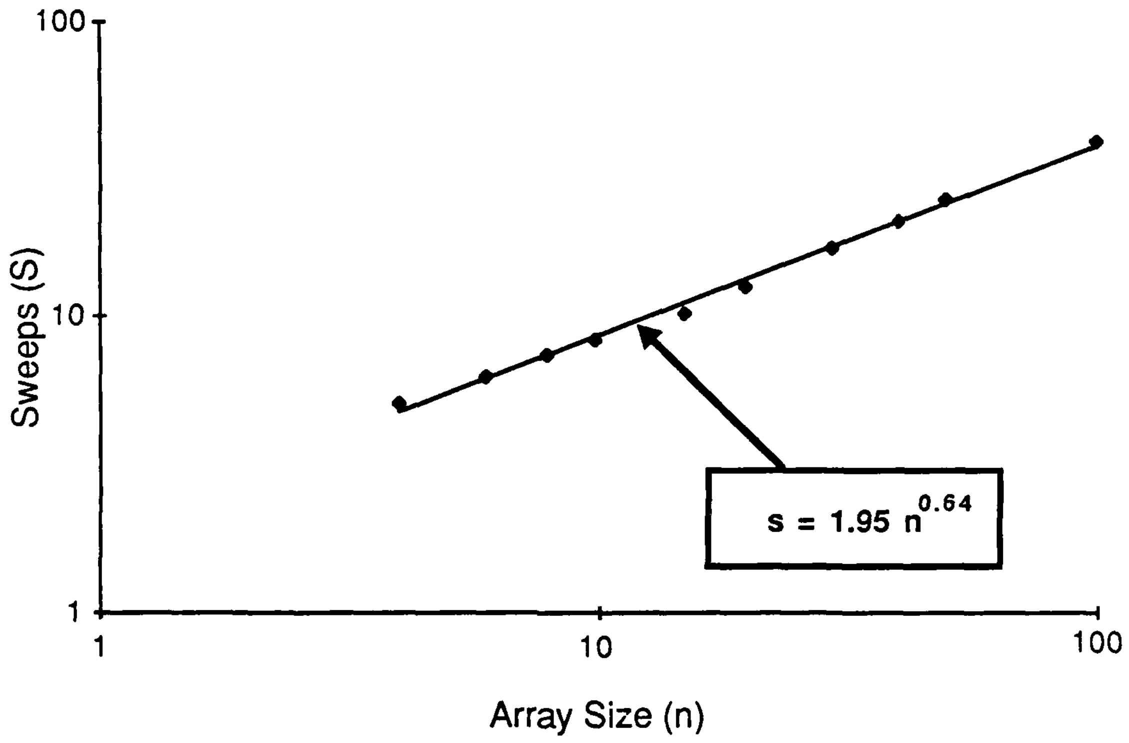
Figure 9.2.1: Sweeps required by the Hestenes Algorithm



n	Avg. Sweeps
4	2.97
6	3.73
8	4.19
10	4.51
20	5.50
30	6.03
40	6.36
50	6.66

Source [Bre85a]

Figure 9.2.2: Sweeps required by the Jacobi Algorithm



n	Avg. Sweeps
4	5.04
6	6.29
8	7.29
10	8.29
15	10.3
20	12.5
30	16.7
40	20.7
50	24.5
100	38.5

Source [Fin83]

Figure 9.2.3: Sweeps required by Finn's approximate Hestenes algorithm (Method C)

9.3 Moreno Pipelined Architecture

9.3.1 Number of Parallel Processors (P) and Stages (S)

Analysis of the number of AUs and computation time for the Moreno architecture is complicated by the fact that it is not just one architecture but is a whole family of architectures. As described in section 8.1, Moreno's design can use several of the processors shown in Figure 8.1.1 in parallel. Additionally the number of stages within each of the processors can be varied to produce a range of throughput rates. The different members of the family are defined by two variables, P and S; where P is the number of parallel processors and S is the number of stages in each processor. There are two constraints on P and S. First, each stage of each processor will be operating on 2 columns of A simultaneously. Therefore, to keep all stages busy 100% of the time, the number of stages (PS) must be less than n/2. (Actually, Moreno shows that in order to resolve dependencies in the column exchange unit $PS \leq n/2 - 1$ [Mor85].) Second, there is an upper bound on S. As shown in Figure 8.1.2, in Moreno's design there are 2 stages in the rotation unit and a maximum of $\lceil \log_2(m) \rceil + 1$ stages in the inner product tree. (In his thesis, Moreno only counts the rotation and inner product unit as 1 stage each. This appears to be an error since each unit is pipelined and can be operating on more than 1 column pair at a time.) The rotation angle and norm update unit can have at most 59 stages since there are 59 nontrivial operations to be performed. Therefore the maximum number of stages in a processor is given by $S \leq 62 + \lceil \log_2(m) \rceil$.

We have chosen to concentrate on the group of designs with P = 1. The single processor design was analyzed since this selection will always yield the most efficient design. As P increases relative to n, the number of stages in each processor must decrease to maintain the relationship $PS \leq n/2 - 1$. As S decreases it becomes more difficult to match the throughput rate of each stage exactly . Therefore some stages will have underutilized processors. The choice of P=1 provides the best opportunity to match the throughput rates of all stages.

Our choice of S is dictated by the goal of having the fastest possible computation time while maintaining reasonable efficiency for the individual AUs. The fastest computation time is achieved when the throughput rate of each stage is one column pair per time step. (i.e. we are able to complete the orthogonalization of two columns every time step.) This rate can only be achieved when S is a maximum ($= 62 + \lceil \log_2(m) \rceil$) and when there are a sufficient number of columns to keep the pipe completely full ($n \geq 2S + 2$). For a square matrix these conditions are satisfied when $n \geq 126+2\lceil \log_2(n) \rceil$. This equation is satisfied in the equality sense for n = 142. Therefore for all values of $n \geq 142$, S will be set to $62 + \lceil \log_2(n) \rceil$. Below n = 142, S will be set to $\approx n/2 - 1$.

9.3.2 Number of AUs and OPs for $n \geq 142$

For $n \geq 142$, the throughput rate of each stage can be set to 1 column pair per time step. In order to support this rate the rotation angle / norm update unit must have 1 AU per operation or a total of 59 AUs. The inner product tree must have n multipliers at the leaves and n adders in the tree or a total of 2n AUs. The rotation application unit must have two multipliers and an adder for each of the 4n values it produces per orthogonalization or a total of 12n AUs. Overall, the total number of AUs required by Moreno's architecture (C_{Mor}) is

$$C_{Mor} = 14n + 59 = O(n) \text{ AUs}, \quad \text{for } n \geq 142 \qquad (9.3.2.1)$$

Since the throughput rate is one orthogonalization per time step the computation time is equal to the total number of orthogonalizations required to compute the SVD. The design performs exacıly n(n-1)/2 orthogonalizations during every sweep. As shown above, the Hestenes algorithm requires $2.7\log_2(n) + 2.0$ sweeps to converge. Therefore the computation time for the Moreno architecture is given by:

$$T_{Mor} = [2.7 \log_{10}(n) + 2.0]\,(n/2)\,(n-1) = O(n^2 \log n) \text{ OPs, for } n \geq 142 \qquad (9.3.2.2)$$

9.3.3 Number of AUs and OPs for n < 142

Below n = 142 we must have S = n/2 -1 and we must allocate AUs to equalize the throughput of each stage. As shown in [Mor85], the number of stages, the throughput rate and the number of AUs for each of the major units in Moreno's design using non-pipelined AUs are as follows:

Unit	Stages	Throughput	# of AUs	Remarks
Rotation Computation	S_θ	$S_\theta/59$	S_θ	Includes norm update
Rotation Application	2	G/4n	3G	G = # of M/M/A units
Inner Product	$\log_2(F)+1$	F/n	2F	F = # of leaves in tree

(Note that the throughput rates shown are only approximately those computed by Moreno. His equations include ceiling functions which make further use of the formulas very difficult. For the purposes of this study the approximate values provide adequate estimates of computation times and numbers of arithmetic units once n is larger than, say, 10.)

To satisfy the requirements on the number of stages and to have equal throughput rates for all stages we must have

$$S_\theta + \log_2(F) + 3 = n/2 - 1 \qquad (9.3.3.1)$$

and

$$S_\theta/59 = G/4n = F/n \qquad (9.3.3.2)$$

From equation (9.3.3.2) we see that $F = S_\theta n/59$. Substituting this expression for F into equation (9.3.3.1) gives

$$S_\theta + \log_2(S_\theta n/59) + 3 = n/2 -1 \qquad (9.3.3.3)$$

or rearranging terms

$$S_\theta + \log_2(S_\theta) - \log_2(59) + 4 = n/2 - \log_2(n) \qquad (9.3.3.4)$$

Noting that $\log_2(59) \approx 6$ we see that we can determine a value for S_θ from

$$S_\theta + \log_2(S_\theta) - 2 \approx n/2 - \log_2(n) \qquad (9.3.3.5)$$

Once S_θ is found, equation 9.3.3.2 can be used to compute values for F and G and from them the number of AUs can be computed. Doing so we find the the total number of AUs (C_{Mor}) is given by

$$\begin{aligned} C_{Mor} &= S_\theta + 2F + 3G = S_\theta + 2\, S_\theta n/59 + 3(4\, S_\theta n/59) \\ &= S_\theta(1 + 14n/59), \quad \text{for } n < 142 \qquad (9.3.3.6) \end{aligned}$$

Since S_θ is O(n) we see that the number of AUs in the Moreno architecture is $O(n^2)$ for $n < 142$ if we wish to maintain high efficiency.

The computation time for $n < 142$ is again the time for an orthogonalization times the number of orthogonalizations required to complete the SVD. In this case the orthogonalization time is the inverse of the throughput for the slowest stage. Since all stages have been designed to have equal throughput rates the orthogonalization time equals $59/S_\theta$. Therefore the total computation time is given by:

$$T_{Mor} = (59/S_\theta)\, [2.7 \log_{10}(n) + 2.0]\, (n/2)\, (n - 1) \quad \text{OPs for} \quad n < 142 \qquad (9.3.3.7)$$

Since S_θ is O(n), the computation time for Moreno's architecture is O(nlogn) for $n<142$. However the proportionality constant is very high.

9.4 Schimmel/Luk Linear Array

9.4.1 Arithmetic Units

The first unit we will consider in the Schimmel/Luk array is the matrix multiplier. In order to achieve a fast computation time for H (= UΣ) and V, the rotations should be applied to both matrices in parallel. This requires the matrix multiplication unit to have 2n columns. The original Schimmel/Luk design has three rows of processors in the matrix multiplication unit. However, on close analysis of the structure of the tridiagonal matrix it can be seen that every other element of the sub and super diagonals is zero. If this structure is exploited the number of rows in the matrix multiplication unit can be reduced to two [Sch86b]. The top row requires only one AU per processor to compute a multiplication. The bottom row will require two AUs per processor, one for a multiplication and one for an addition. Hence the matrix multiplier unit needs 6n AU's.

The inner product unit as shown in Figure 8.2.2.2 would require 3n AUs, 2n for the top row and n for the bottom. The purpose of the top row is to compute the norm of each column. An alternate method is to compute the norms once at the beginning of the algorithm and then update them during each sweep to reflect the effects of the rotations. If columns i and j are being orthogonalized then the updated norms of columns i and j (ρ_i' and ρ_j', respectively) are given by

$$\begin{bmatrix} \rho_i' \\ \rho_j' \end{bmatrix} = \begin{bmatrix} \cos^2\theta & \sin^2\theta \\ \sin^2\theta & \cos^2\theta \end{bmatrix} \begin{bmatrix} \rho_i \\ \rho_j \end{bmatrix} + \begin{bmatrix} -2\gamma_{ij}\cos\theta\sin\theta \\ 2\gamma_{ij}\cos\theta\sin\theta \end{bmatrix} \qquad (9.4.1.1)$$

where ρ_i and ρ_j are the original column norms, γ_{ij} is the inner product of columns i and j and θ is the rotation angle. If this norm update procedure is performed in the rotation*solver unit, then the top row of cells in the inner product unit can be eliminated. This reduces the number of AUs in the inner product unit to n.

The final section of the Schimmel/Luk design which must be considered is the rotation solver unit. The performance of this unit is critical. To support the matrix multiplication unit it must produce a new set of rotation parameters (cos θ and sin θ) every other time step. This suggests that the unit be pipelined with a throughput rate of one rotation computation per 2 time steps. Figure 9.4.1.1 shows the dependency graph of the rotation computation. The figure also shows the dependency graph for the norm update computation since this calculation has been moved to the rotation solver. A count of the number of non-trivial operations (multiplications by 2 and additions of 1 are ignored) gives a total of 59 OPS (allowing 9 OPs for divisions and 12 for square-roots). Therefore if approximately 30 AUs were used in the rotation solver pipeline, the desired throughput rate of one rotation every two OPs could be attained.

However, there is a problem with this assignment of AUs; the delay through the rotation solver. With a pipeline of 30 AUs each performing two OPS the total delay is 60 OPs. This is much too long. The start of each matrix product would have to be delayed by that amount from the time when the first inner product was available from the previous matrix product. This is disastrous for small matrices. For example, suppose that a square matrix with n = 60 is being processed. In a typical iteration, the matrix multiplier will complete its operations on the matrix after 62 OPs. The inner product unit will produce the first inner product one OP later. But then we must wait 60 OPs for the rotation solver to produce the first set of rotation parameters. During those 60 OPS the 6n (= 240) AUs in the matrix multiplier are sitting idle. The AUs in the matrix multiplier (and those in the inner product unit as well) will only be busy approximately half of the time. This problem is even worse for matrices with n smaller than 60. It does become less

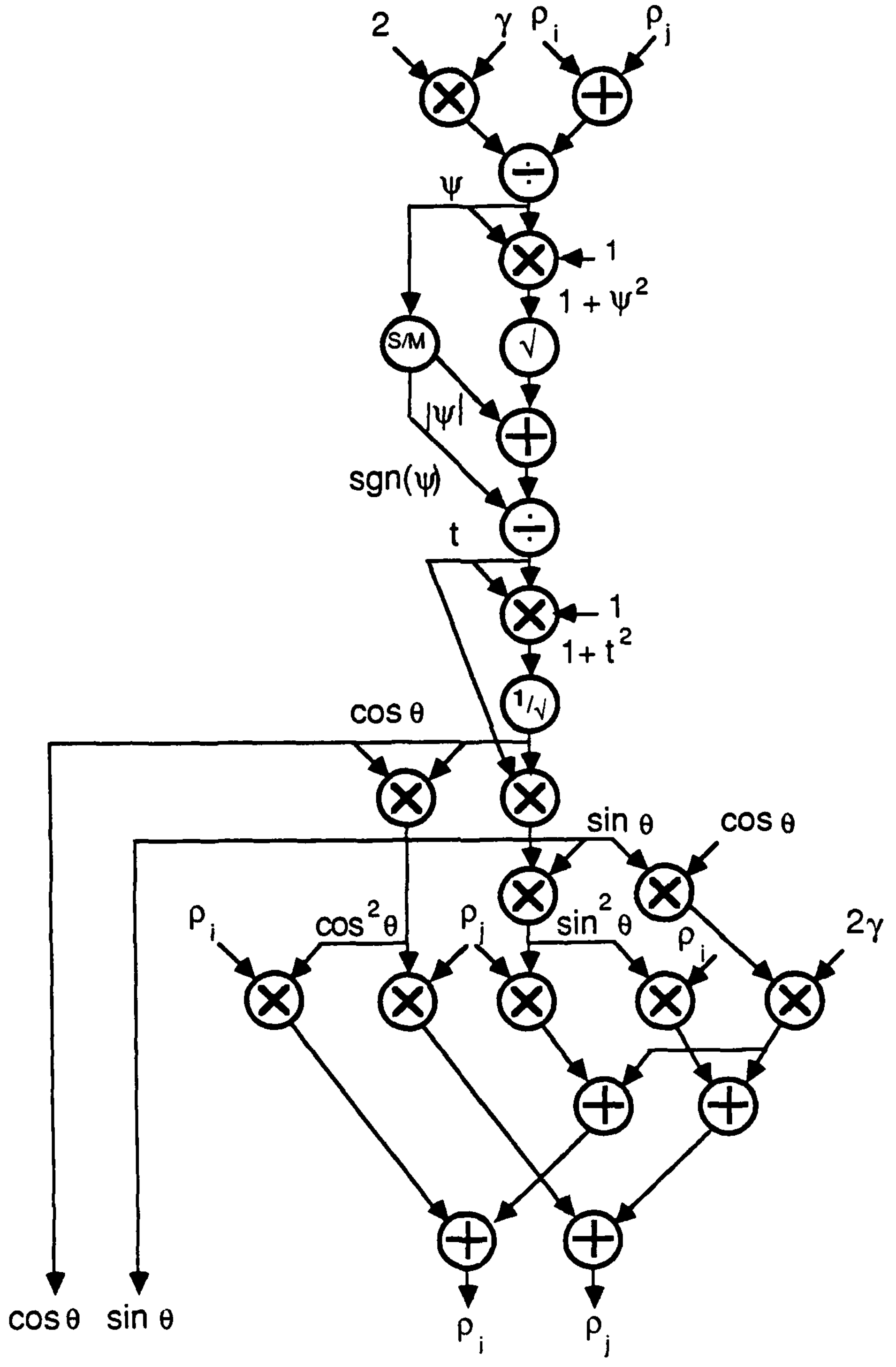


Figure 9.4.1.1: Dependency graph for the Rotation angle computation and Norm update in the Schimmel/Luk SVD architecture

severe for square matrices with n greater than 60 or for matrices which have many more rows than columns.

The delay through the rotation solver can be reduced somewhat by recognizing that the norm update computation can be accomplished in parallel with the matrix multiplication. That is, as soon as the rotation parameters are available they can be transmitted to the matrix multiplier and simultaneously to a norm update processor. Analysis of Figure 9.4.1.1 reveals that the rotation parameters are available after 47 OPs. However, this is still a long delay.

This problem was discussed with Schimmel and Luk. They suggested that the delay could be reduced by speeding up the division and square root computations. They noted that the number of iterations required for these operations can be reduced by improving the initial "guesses". This can be done by increasing the number of entries in the table which contains the initial values. With a sufficiently large look-up table an adequate division or square root result can be obtained in one iteration. This would translate to 4 OPs for a division and 5 for a square root and would reduce the overall rotation calculation to 23 OPs. If a pipeline of 23 AUs are used in the rotation solver connected in a manner which exploits the limited parallelism in the rotation calculation, the delay through the unit is reduced to approximately 20 OPs. This pipeline of 23 AUs was the configuration finally accepted.

The rotation unit also requires AUs for the norm update. Note that after the first set of rotation parameters is delivered, the rotation unit can compute the following sets at a rate of one every other time step. Therefore the norm update unit can operate at half the rate of the rotation solver. Since there are 12 OPs in the norm update, only 6 AUs are required. This brings the total AUs in the rotation unit to 29.

In summary the Schimmel/Luk array requires 6n AUs for the matrix multiplier, n for the inner product unit and 29 for the rotation solver.

The total ($C_{Sck/Luk}$) is given by

$$C_{Sck/Luk} = 7n + 29 \quad = O(n) \text{ AUs} \tag{9.4.1.2}$$

9.4.2 Computation Time

The Schimmel/Luk array uses the odd-even ordering scheme shown in section 2.2 to generate rotation pairs. Therefore the design requires the data matrix to flow through the array n times for each sweep in order to perform all n(n-1)/2 possible column pair orthogonalizations. A total of n+3 OPs are required for the first row of the matrix to be processed by the matrix multiplier and inner product unit. An additional 20 OPs are required for the delay through the rotation solver. At that point the first row can be processed again. Therefore each data pass requires n + 23 OPs of time. Accordingly each sweep requires n(n + 23) OPs. Finally, since the Hestenes algorithm requires $2.7 \log_{10}(n) + 2.0$ sweeps, the total computation time ($T_{Sch/Luk}$) is given by

$$T_{Sch/Luk} = [2.7 \log_{10}(n) + 2.0]\, n\, (n + 23) = O(n^2 \log n) \quad \text{OPs} \tag{9.4.2.1}$$

9.5 Brent/Luk/Van Loan (BLV) Square Array

9.5.1 Rotation Parameter Computation

In the two previous architectures, we have seen that the computationally complex calculation of the rotation parameters is performed by a highly pipelined, special purpose unit. This was possible since the architectures were designed to keep such units busy. However, in the case of the BLV array and the other two quadratic arrays which follow, it is not possible to pipeline the rotation computations. In these arrays some (or possibly all) of the processors must compute and apply rotation parameters and then transmit them to neighboring cells. These processors must then wait for their neighboring cells to apply the rotations. Only then can they exchange data with their neighbors and start on the next set of rotation parameters. Therefore, pipelining can not be used. Accordingly, we can use only a few AUs in the rotation parameter computation effectively. Since the number of AUs is limited, the computation time for the rotation parameters will be long.

9.5.2 Arithmetic Units

In the BLV array the diagonal processors compute rotations and apply them. Therefore the diagonal processors should be designed to take advantage of even the small amount of parallelism available in the rotation computation. A dependency graph for the rotation parameter computation in the Jacobi algorithm is shown in Figure 9.5.2.1. Recalling that the iterative algorithms for divisions and square-roots can use two AUs in parallel, the figure shows that 4 AUs can be used effectively in the computation. In their paper [Bre85a], Brent, Luk and Van Loan recommended the use of an alternate computation procedure for the

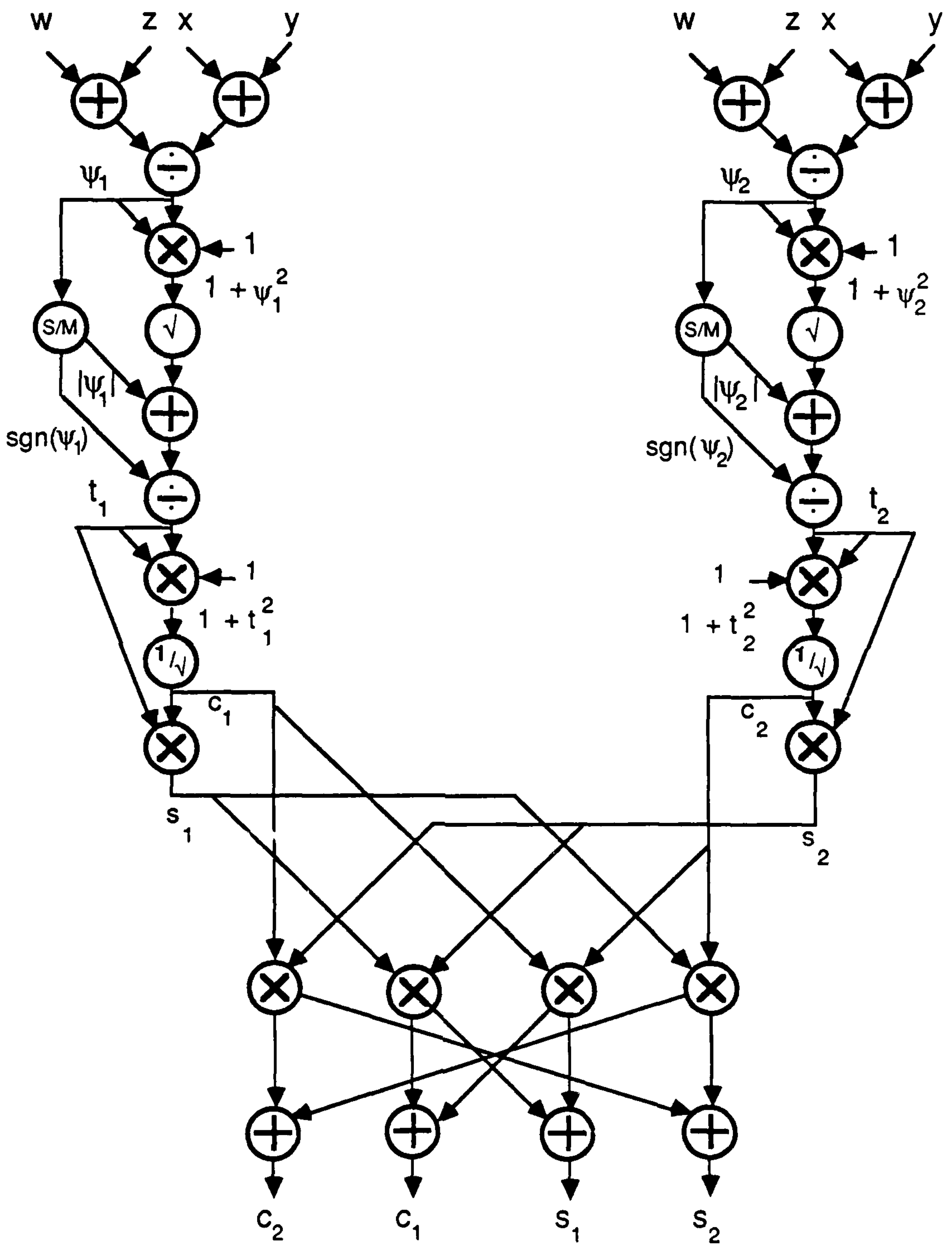


Figure 9.5.2.1: Dependency graph for algorithm FHSVD

rotation parameters. They call it algorithm USVD and a dependency graph for it appears in Figure 9.5.2.2. They prefer algorithm USVD because it only has 3 divisions and 3 square roots as opposed to the 4 divisions and 4 square roots in algorithm FHSVD which appears in Figure 9.5.2.1. However the dependency graph for algorithm USVD shows much less parallelism than that of algorithm FHSVD. Therefore in the subsequent analysis it will be assumed that algorithm FHSVD is used and 4 AUs will be allocated to each diagonal element.

The off-diagonal processors have many fewer operations to perform. In fact, they spend most of their time waiting for the diagonal processors. Therefore only one AU will be assigned to each off-diagonal processor.

Since there are a total of $n^2/4$ processors in the BLV array with n/2 of them on the diagonal the total number of AUs (C_{BLV}) required is given by

$$C_{BLV} = n^2/4 + 3n/2 = O(n^2) \text{ AUs} \qquad (9.5.2.1).$$

9.5.3 Computation Time

Brent, Luk and Van Loan state in their paper that the processors (and the communication links) in their array are busy only one third of the time [Bre85a]. This is a somewhat pessimistic conclusion. We believe that the two time periods between the normal active periods could be used for the computation of the U and V matrices. This could be done by having each processor retain a copy of the rotation parameters for three time periods. During the first period of a three period cycle, a processor would: accept (or compute) new rotation parameters; apply them to the A matrix; transmit the parameters to its neighbors; and exchange elements of the U matrix (completed during the previous three cycle period) with its neighbors. In period two, the processor would aoply the rotations to the U matrix and exchange elements of the V matrix (completed during the

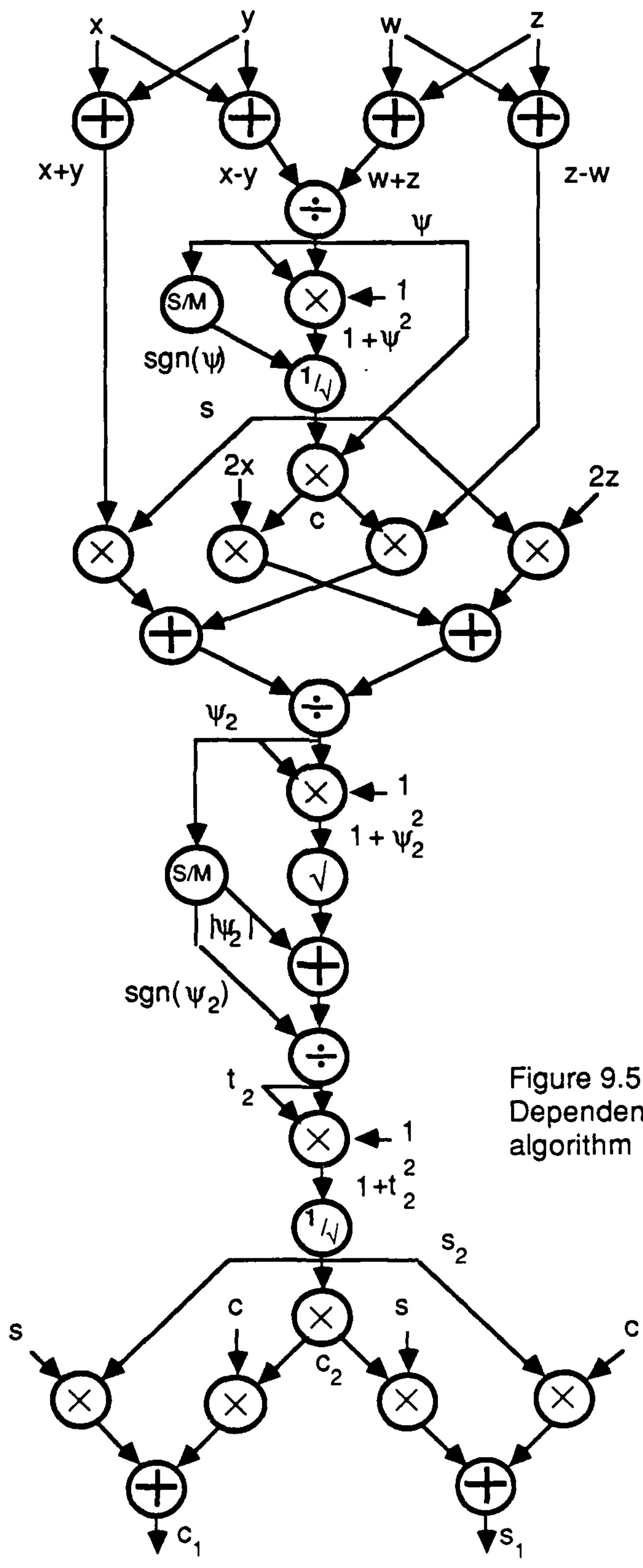


Figure 9.5.2.2: Dependency graph for algorithm USVD

previous three cycle period) with its neighbors. During period three, the processor would apply the rotations to the V matrix and exchange elements of the A matrix with its neighbors. In this way the U and V matrices could be computed in the BLV array with no increase in time over that of Σ alone. In the SVD time calculation for the BLV array it has been assumed that this savings will be exploited.

The computation time for Σ in the BLV is found as a product of the number of sweeps, the number of iterations per sweep required to annihilate all off-diagonal elements, and the time for each iteration. The number of sweeps (as shown in Section 9.2) is equal to $3.1 \log_{10} n + 1.5$. Since each of the n/2 diagonal processor annihilates 2 off-diagonal elements during each iteration, a total of n of the off-diagonal elements are annihilated per time step. Therefore at least n-1 iterations are required to annihilate all n(n-1) off-diagonal elements. The BLV array achieves this minimum since it uses a round robin communication pattern. The structure of the BLV array allows more than one iteration to be processed simultaneously. In fact a new iteration is started each time the diagonal processors compute rotations. Therefore the time for an iteration is the time required for the diagonal processors to complete the three period cycle described in the previous paragraph. In order to compute the time for the three period cycle, an assumption must be made on the timing strategy used to synchronize the processors in the array. The two possibilities are a systolic architecture or a data flow design.

If the array is assumed to be systolic then there must be a global clock which provides a periodic timing signal to control data transfers between processors. The clock period would have to be longer than the time for the longest computation, which is the time for the computation of rotations (t_r). The time for the three period cycle for each iteration would be at least $3t_r$ for the systolic design. Using 4 AUs in a diagonal element, t_r is 41 OPs and the iteration time is 123 OPs.

Alternatively, the array could be based on a data flow design. In this scenario each individual processor is self-timed and performs its functions as soon as all necessary operands are available. Data transfers are controlled by "handshaking data registers". If the BLV array used a data flow structure, the overall computation time would be dictated by the time for the slowest processors, the diagonal elements. In a typical three period cycle a diagonal element (with 4 AUs) requires 41 OPs to compute and apply rotations. Then it must wait for the next two time periods while the off-diagonal processors apply rotations. The computation performed by the off-diagonal processors (shown in Figure 8.3.1) requires 24 OPs with 1 AU. Therefore the diagonal processor must wait for 48 OPs before its next set of operands are available. (Actually, as described above, the diagonal processor can be applying the rotations to its portion of the U and V matrices during the wait time.) As a result the time for an iteration for the data flow design equals 41 + 48 = 89 OPs. Note that it is possible to lower this time somewhat by using more than one processor in the off-diagonal elements. For example if two AUs were used, the application of rotations would take only 12 OPs and the iteration time would be 65 OPs. However adding one AU to each off-diagonal processor would double the area consumed by the array. This doubling of the area gives only a 27% decrease in the computation time. So unless speed is of paramount importance, the use of more than 1 AU per off-diagonal processor does not appear to be cost effective.

In summary, the data flow design provides a faster iteration time than the systolic design (89 versus 123 OPs). The data flow design will be used for all further computations and comparisons. With an iteration time of 89 OPs the SVD time the for the BLV array (T_{BLV}) is given by

$$T_{BLV} = 89\,[3.1 \log_{10}(n) + 1.5]\,(n-1) = O(n\log n) \text{ OPs} \qquad (9.5.3.1)$$

If a constant number of sweeps (say 10) are used then the computation time for the BLV array is O(n). However, the proportionality constant is almost 900. For the linear arrays described above, the computation time is $O(n^2)$ if a constant number of sweeps are used. But the proportionality constants are in the range of 5 to 10. This large disparity in the proportionality constants will become important in the comparison of the architectures given in a later chapter.

9.6 Luk Triangular Array

9.6.1 General Comments

The analysis of the Luk triangular array is very similar to that of the BLV array in the previous section. Again we have processors on the main diagonal which compute and apply rotations and off-diagonal processors which apply rotations. There are some minor differences in that the Luk array uses a different interconnection pattern and rotation ordering scheme. The major difference is that the Luk array operates on a upper triangular matrix (R) instead of a full data matrix. It must compute this matrix prior to the SVD computation, so the time required for the QR decomposition must be included in the overall computation time. Like the BLV array, each processor in the Luk array is only active for a fraction of the total computation time. In the Luk array each processor is active for one period out of a four period cycle.

The Luk array is ideally suited for the computation of Σ only. It can be extended to compute U and V by adding additional processors. The number of AUs is computed below for both cases. As in the BLV case, the computation of U and V can be done in a manner which does not increase the computation time. That is, the processors can apply rotations to the U and V matrices during their normally inactive periods.

9.6.2 Arithmetic Units

The Luk triangular array has $\lceil n^2/4 \rceil - 1$ ($\approx n^2/4$) off-diagonal processors if we are computing Σ alone or $n^2/2$ if we want Σ, U and V. Following the rationale given for the BLV array, we would assign one AU to each off-diagonal cell. However, we will see that that we can handle all of the operations of 2 off-diagonal cells with a single AU with no increase in SVD computation time. We can do this because each off-diagonal cell is busy for only 2 time steps out of each 4 step cycle, if we are computing Σ, U and V. Therefore the number of AUs in the off-diagonal processors is $\approx n^2/8$ if we are computing Σ alone or $n^2/4$ if we are computing Σ, U and V.

The Luk array has n-1 diagonal processors. They perform the identical function as the diagonal processors in the BLV array, computing rotations to diagonalize a 2-by-2 submatrix. There is one minor difference, however. Luk uses "outer rotations" as opposed to the "inner rotations" used in the BLV array. He does this because the outer rotations not only diagonalize the submatrix but perform a permutation of the elements as well [Luk86]. This permutation is exactly the one required by the odd-even ordering scheme used in the Luk array. Therefore the exchange of data elements is accomplished as a side-effect of the application of rotations rather than as a separate step. Luk recommends a two step procedure similar to algorithm USVD shown in Figure 9.5.2.2 for the computation of the rotations [Luk86]. Again for improved computation time, it appears that an algorithm such as FHSVD shown in Figure 9.5.2.1 modified to compute outer rotations would be preferable. This algorithm can use 4 AUs per diagonal processor. As in the case of the off-diagonal cells we can handle the functions of two of the diagonal cells with one set of AUs. Accordingly we need a total of 2(n-1) AUs for the diagonal cells,

In summary, the total number of AUs for the Luk array (C_{Luk}) is given by

$$C_{Luk} = \begin{cases} n^2/8 + 2(n-1) \text{ to compute } \Sigma \text{ alone} \\ n^2/4 + 2(n-1) \text{ to compute } U, \Sigma \text{ and } V \end{cases} \qquad (9.6.2.1)$$

9.6.3 Computation Time

The analysis of the computation time for the Luk array is similar to that of the BLV array. Again it will be assumed that a data flow architecture is used. We must however account for the time required for the QR decomposition (QRD) at the start of the algorithm.

The computations performed in the QRD are similar to those in the SVD. For the QRD, the diagonal processors compute a single set of rotation parameters to annihilate sub-diagonal elements and transmit them across the rows of the array. The off-diagonal cells apply the rotations. The process continues until all n(n-1)/2 elements of the lower triangular portion of A are annihilated. From this description we see that the QRD is very similar to one sweep of the SVD computation on the Luk array (except only single rotations are involved rather than the pair of rotations used in the SVD). Therefore for simplicity it will be assumed that the QRD takes half of the time of one of the SVD sweeps.

Luk only gives data on the number of sweeps required for convergence of his algorithm for very small matrices ($n \leq 20$). Therefore it will be assumed that it requires the same number of sweeps as that of the BLV array. That is the number of sweeps, including the extra time for the QRD, is $3.1\log_{10}(n) + 2.0$.

During each sweep $n(n-1)/2$ elements must be annihilated in the diagonal processors. As shown in section 2.2 this requires a total of n iterations for the odd-even rotation ordering scheme. In the Luk array each iteration requires one step to compute rotation parameters in the diagonal processors followed by a step to apply rotation parameters in the off-diagonal processors. The following diagram illustrates the activity pattern of the diagonal processors during a single sweep for n=8. The notation (i, j) denotes the annihilation of element r_{ij}. A dash indicates an inactive time period for the processor.

Diagonal Processor	1	2	3	4	5	6	7
Step 1	(1,2)	-	(3,4)	-	(5,6)	-	(7,8)
Step 2	-	-	-	-	-	-	-
Step 3	-	(1,4)	-	(3,6)	-	(5,8)	-
Step 4	-	-	-	-	-	-	-
Step 5	(2,4)	-	(1,6)	-	(3,8)	-	(5,7)
Step 6	-	-	-	-	-	-	-
Step 7	-	(2,6)	-	(1,8)	-	(3,7)	-
Step 8	-	-	-	-	-	-	-
Step 9	(4,6)	-	(2,8)	-	(1,7)	-	(3,5)
Step 10	-	-	-	-	-	-	-
Step 11	-	(4,8)	-	(2,7)	-	(1,5)	-
Step 12	-	-	-	-	-	-	-
Step 13	(6,8)	-	(4,7)	-	(2,5)	-	(1,3)
Step 14	-	-	-	-	-	-	-
Step 15	-	(6,7)	-	(4,5)	-	(2,3)	-
Step 16	-	-	-	-	-	-	-

This diagram clearly illustrates how we can share AUs between cells. For example we see that the functions of diagonal cells 1, and 2 can be handled by a single set of AUs since cells 1 and 2 are never active at the same time. A similar diagram of the off-diagonal cells would show the exact same pattern. We also see that the diagonal cells can compute U and V during their normally inactive "even" time steps.

The diagram also shows that the time for a sweep is n times the sum of the time for an odd numbered step and the time for an even numbered step. During odd numbered steps the rotation parameters are computed using algorithm FHSVD modified to compute outer rotations. As shown in [Luk86], to compute outer rotations we need only replace the formula

$$t = \frac{\mathrm{sign}(\rho)}{|\rho| + \sqrt{1+\rho^2}} \tag{9.6.3.1}$$

by

$$t = -\mathrm{sign}(\rho)\,[\,|\rho| + \sqrt{1+\rho^2}\,] \tag{9.6.3.2}$$

From this we see that the modified version of FHSVD will have one less division than the original. As a result the computation time for the rotation parameters using 4 AUs in the diagonal processors will be 35 OPs (41 OPs - 6 OPs for a division). The time required for the even numbered steps is just the time for 1 AU to apply a set of rotations = 24 OPs. Therefore a sweep requires 59n OPs and the overall SVD computation time for the Luk triangular array is given by

$$T_{Luk} = 59\,[3.1\,\log_{10}(n) + 2.0]\,n \;=\; O(n\log n) \quad \text{OPs} \tag{9.6.3.3}$$

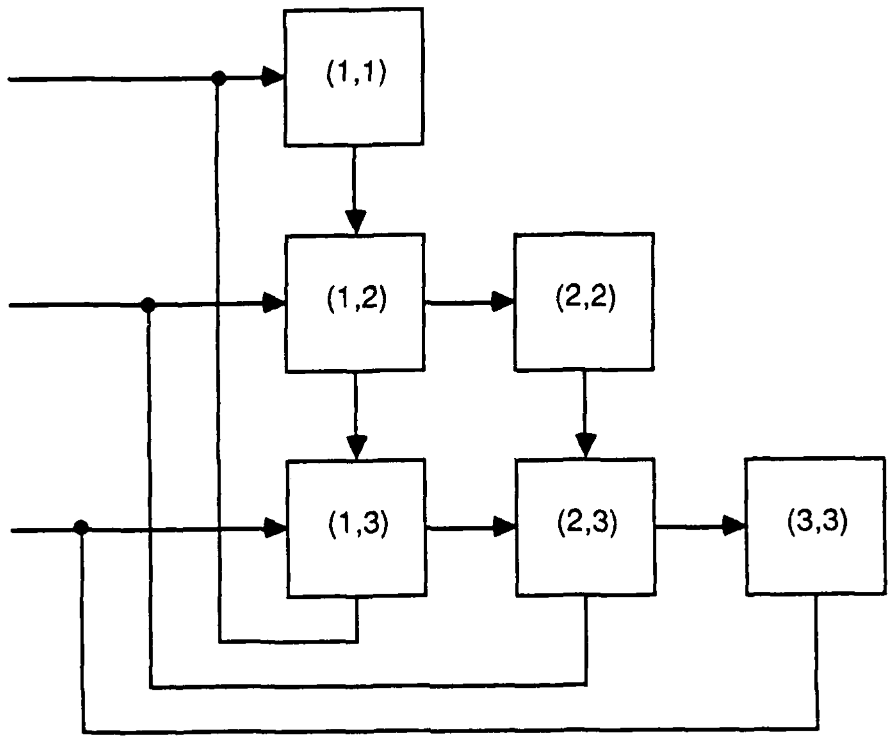
As in the case of the BLV array, if we use a fixed number of sweeps (10) then the computation time is O(n), but the proportionality constant is high ($\approx$ 600).

9.7 Finn Triangular Array

9.7.1 General Data Flow

As shown in section 8.5, the Finn array implements an approximate version of the Hestenes SVD algorithm. To give a better feeling for the data flow in the Finn array, Figure 9.7.1.1 depicts the array structure and the initial data entry for a 3-by-3 matrix. Figures 9.7.1.2 and 9.7.1.3 show the start up phase and a single sweep of the SVD computation. During each time period a processor is either computing an inner product (IP), computing rotation parameters and updating norms (θ/NU), applying rotations (Rot.) or is idle (Wait). The start up phase is required to compute the column norms of the original data matrix and the inner products of the columns. The norms are computed in the first column of processors as the matrix elements enter the array for the first time. After that the norms are updated. Once this start up phase is completed the array performs a number of sweeps (shown in Figure 9.7.1.3) each of which consists of two passes of the data. A data pass is defined to start when element a_{11} enters the (1,1) processor. During the first pass inner products are computed. During the second pass rotations are computed and applied.

It should be noted that the only function of the "diagonal" processors in the Finn array is to store and transmit data values. Therefore they do not require AUs but can be constructed with data registers. The only exception to this is the (1,1) processor which is required to compute the norm of column 1 during the start up phase. After that it is idle. By shifting this norm computation into the (1,2) processor, the (1,1) processor could also be replaced by data registers.



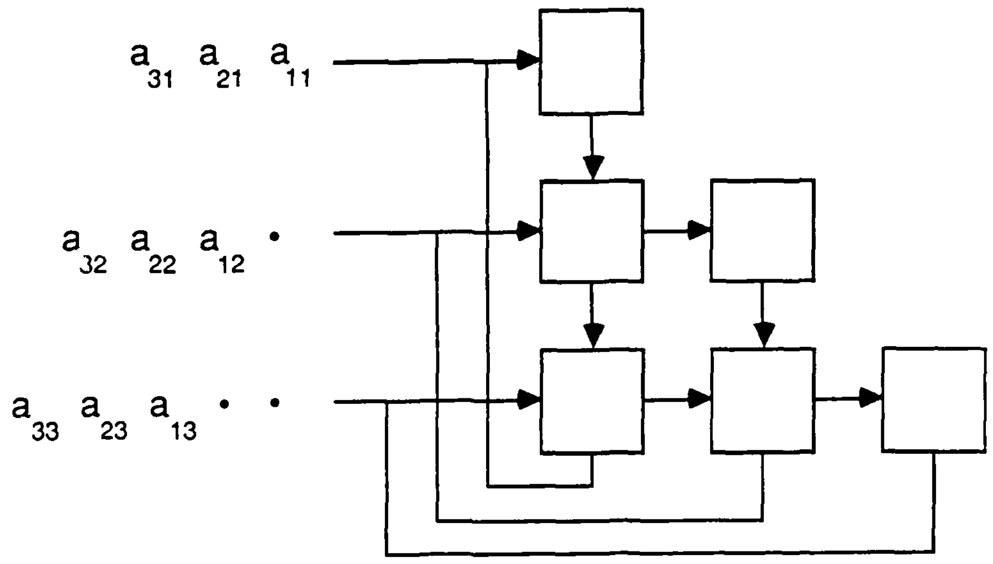

a_{31} a_{21} a_{11}
a_{32} a_{22} a_{12} •
a_{33} a_{23} a_{13} • •

Figure 9.7.1.1: Array structure and initial data entry for the Finn architecture (for a 3-by-3 matrix)

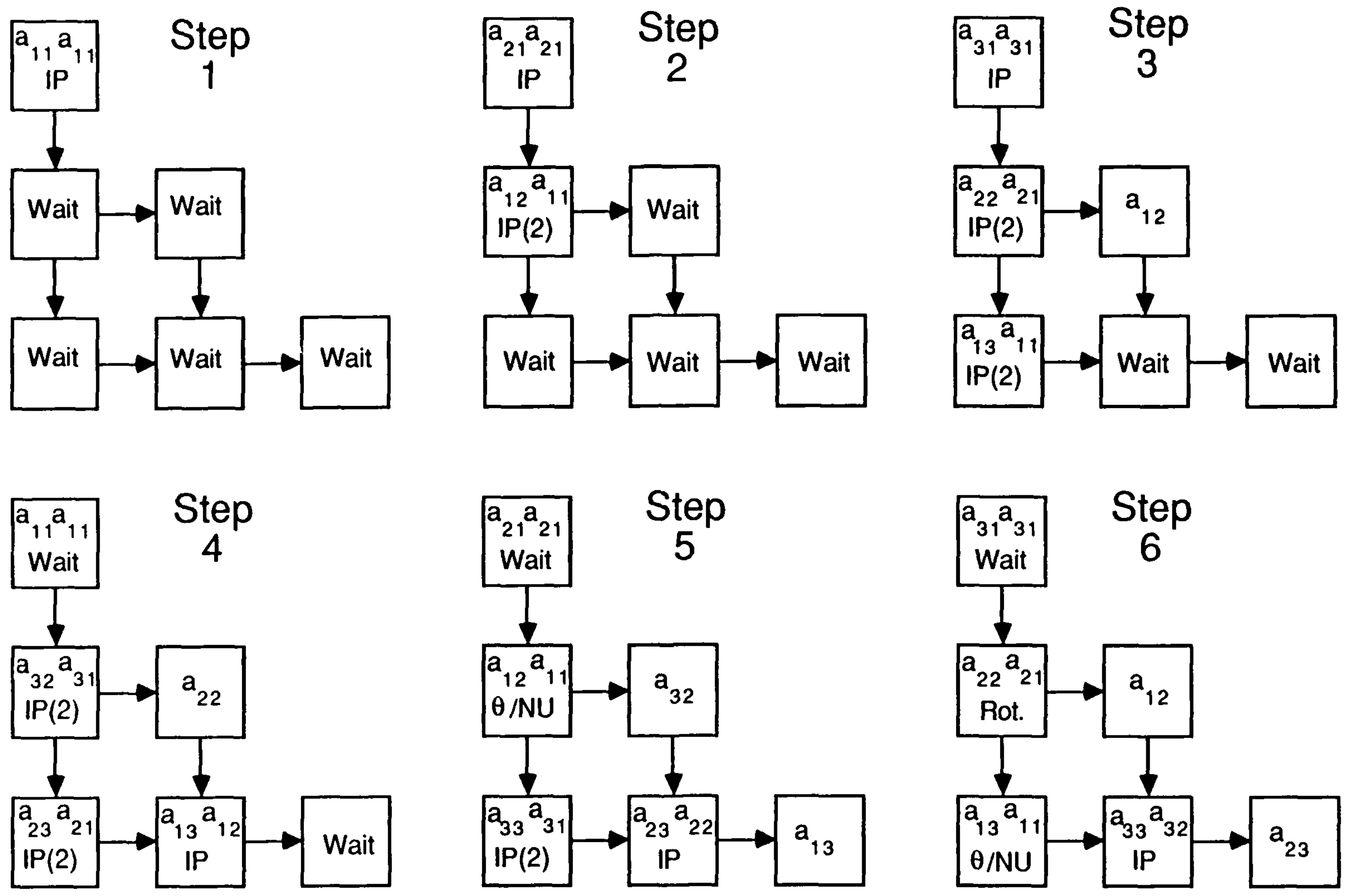


Figure 9.7.1.2: Start-up phase of the SVD computation in the Finn array

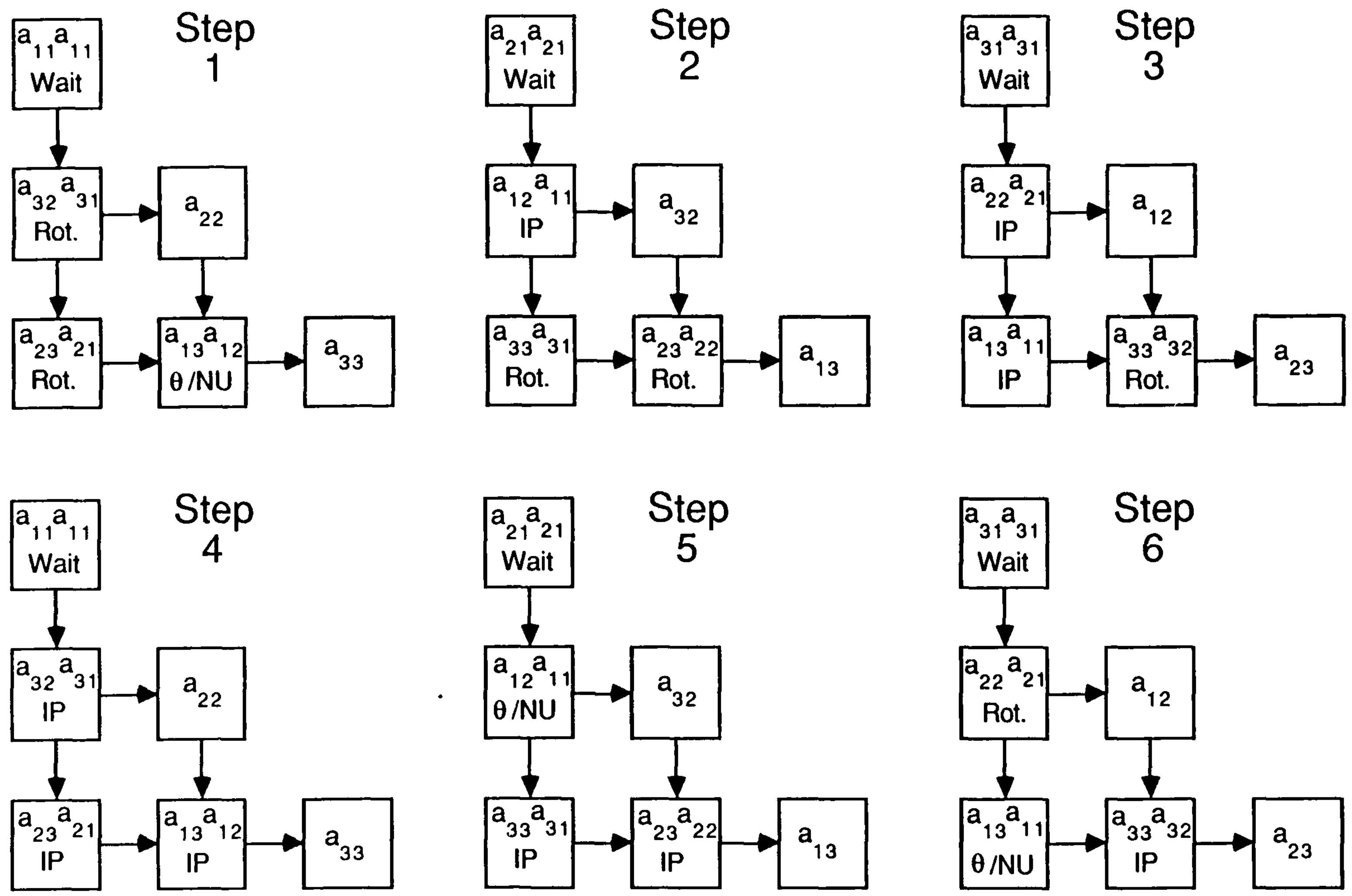


Figure 9.7.1.3: One sweep of the SVD computation in the Finn array

9.7.2 Arithmetic Units

Without the diagonal processors, the Finn array requires n(n-1)/2 processors. The processors are all functionally identical. As usual the number of AUs for each processor is dictated by the rotation/norm update calculation. The dependency graph for this computation is shown in Figure 9.7.2.1. Note that this is the graph for the best approximate method developed by Finn, which he calls Method C. This method provided the fastest convergence [Fin83]. With one AU, 64 OPs are required to complete the computation. This time can be reduced to 45 OPs by using 2 AUs. Adding a third AU only reduces the time to 41 OPs. Accordingly each processor will be allocated two AUs to give a total (C_{Finn}) of

$$C_{Finn} = n(n-1) \approx O(n^2) \text{ AUs} \qquad (9.7.2.1)$$

9.7.3 Computation Time

As shown in section 9.2 the number of sweeps for Method C is equal to $1.95n^{0.64}$ for an n-by-n array. Each sweep in turn requires 2n steps. To compute the time for the 2n steps we will again assume that a data flow structure is used. A careful analysis of Figure 9.7.1.3 shows that during n of the steps at least one element of the array is computing rotations, updating norms and applying the rotations to two of the data elements. With two AUs, each of these steps requires 48 OPs (45 for the θ/NU computation and 3 to apply the rotation). During the remaining n steps all processors are applying rotations or computing inner products. With two AUs, 3 OPs are required to apply a rotation while only 2 OPs are required for the inner product computation. Therefore the total time for the 2n steps in a sweep is 51n OPs and the total computation time is given by

$$T_{Finn} = 51n\,(1.95\, n^{0.64}) \approx 100\, n^{1.64} = O(\, n^{1.64}) \text{ OPs} \qquad (9.7.3.1)$$

This rate of growth is much higher than that of the other quadratic arrays.

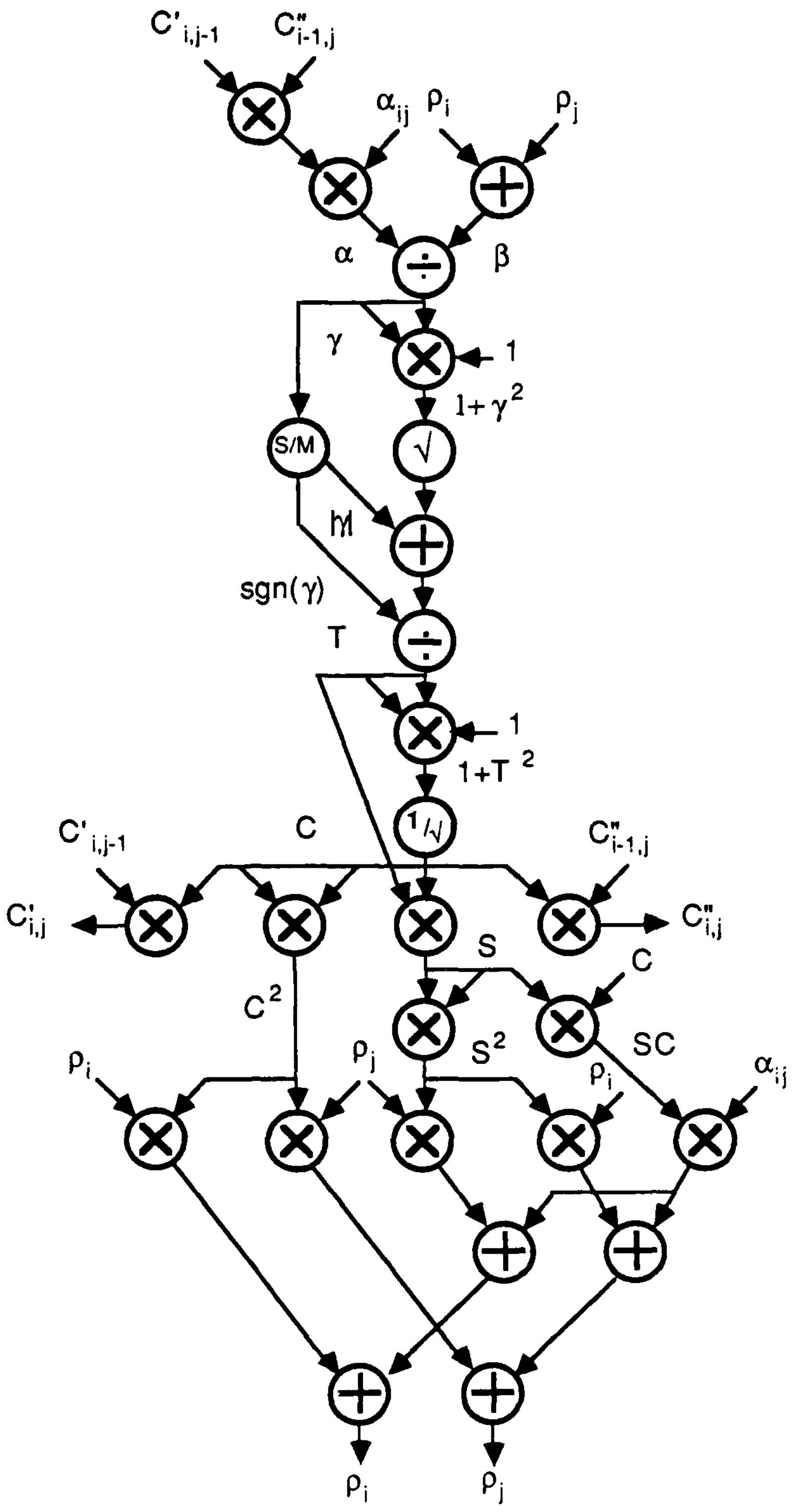


Figure 9.7.2.1: Dependency graph for the rotation angle/norm update computation of Finn's Method C

10.0 COMPARISON OF SVD ARCHITECTURES WITH FLOATING POINT AUs

The purpose of this chapter is to compare the resource requirements of each of the SVD architectures described in the previous two chapters. Table 10.1 summarizes the data presented in Chapters 8 and 9 for each of the architectures. The last row of the table contains entries for a category called AUs x OPs. This is just the product of the number of AUs and the computation time for each architecture. This category is provided as an attempt to quantify the total resource requirements of each architecture and to permit comparison between the linear and quadratic designs. It is similar to the Area-Time metric used in VLSI complexity theory. The table shows that all of the architectures except Finn's have an AU x OP product that is $O(n^3 \log n)$. Finn's is larger, $O(n^{3.64})$, due to the higher rate of growth in the number of sweeps for his approximate algorithm. Asymptotically all of the other four designs have similar resource requirements. However we will see that some significant differences emerge when time and area are considered separately and when the specific proportionality constants associated with each architecture are included in the comparison.

The comparison of the architectures will be presented as a series of charts which show the number of AUs and OPs as a function of n, the size of the data matrix. Normally two charts are presented for each comparison. The first shows the characteristics of the architectures for large values of n (up to n = 1000). This gives the asymptotic behavior of the designs. The second chart gives the data for small values of n (up to n = 100). These charts are included because hardware technology available now and in the immediate future will not support the construction of SVD arrays for large values of n. As of this date a single AU (in particular a parallel floating point multiplier) consumes the majority of an

Table 10.1
COMPARISON OF SVD ARCHITECTURES
(Computing U, Σ and V of an n-by-n matrix)

	MORENO	SCHIMMEL LUK	BRENT, LUK VAN LOAN	LUK	FINN
STRUCTURE	*Pipelined*	*Linear* Array	*Square* Array	*Triangular* Array	*Triangular* Array
ALGORITHM	*Hestenes*	*Hestenes*	Jacobi	Jacobi	Approximate Hestenes
ROTATION ORDER	Round-robin	Odd-even	Round-robin	Odd-even	Sequential
SWEEPS/SVD	2.7log(n)+2	2.7log(n)+2	3.1log(n)+1.5	3.1log(n)+2	$1.95n^{0.64}$
OPS/SWEEP	$n(n-1)/2$, $n \geq 142$ $29.5\, n(n-1)/S_\theta$, $n<142$	n(n+23)	89(n-1)	59n	51n
SVD TIME (OPs)	$O(n^2 \log n)$ $n \geq 142$	$O(n^2 \log n)$	O(nlogn)	O(nlogn)	$O(n^{1.64})$
ARITHMETIC UNITS (AUs)	$14n+59$, $n \geq 142$ $S_\theta(1+14n/59)$, $n<142$	7n+29	$n^2/4+3n/2$	$n^2/4+2(n-1)$	n(n-1)
OPs x AUs	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^{3.64})$

integrated circuit. Therefore initial attempts to construct SVD processors will by necessity be limited to values of $n \leq 100$. The charts for $n \leq 100$ are provided to show the trade-offs between SVD architectures which could be constructed now or in the immediate future using VLSI and possibly wafer scale integration.

In order to provide reasonable graphs of the data, most of the charts are semi-logarithmic. It is important to remember this since the display format tends to mask significant differences between architectures. For example a factor of two difference in the computation time appears as a small, constant, vertical displacement between two curves in the graphs.

10.1 Total Number of Computations Required

Each of the architectures implements slightly different versions of one of the two SVD algorithms. This immediately gives rise to some differences in the total number of computations required to compute the SVD. Figures 10.1.1 and 10.1.2 are provided to show these differences. The charts show the number of operations that would be performed by a single AU to complete an SVD using each of the specific algorithms (we will denote this quantity by $T^{(1)}$) the computation time for a 1 AU system.

The first chart (10.1.1) shows the number of operations required by the basic Hestenes and Jacobi algorithms. It also shows the number of operations required by the Golub-Reinsch algorithm. These curves were developed by a careful count of the number of floating point adds and multiplies (recall that an add and a multiply are both considered to be one OP) for each algorithm. For the Golub-Reinsch algorithm it was assumed that three QR iterations would be required to reach convergence for each singular value. For the Hestenes and Jacobi algorithms it was assumed that the number of sweeps for convergence is

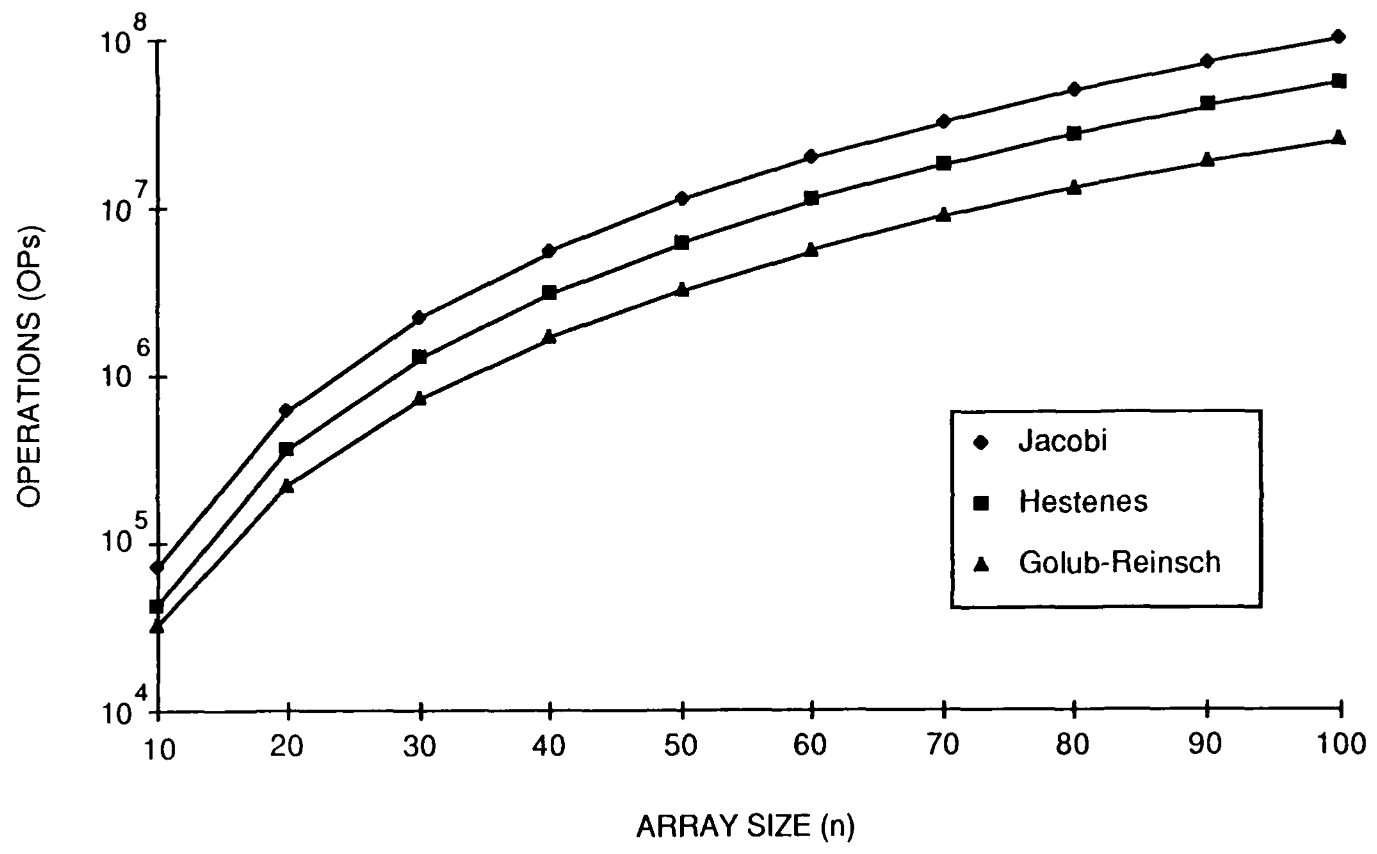


Figure 10.1.1: Total operations required by different SVD algorithms to compute U, Σ and V for square (n-by-n) matrices.

O(logn) with the exact constants given by the formulas developed in Section 9.2. With these assumptions the following formulas were developed to relate the total number of OPs required to compute U, Σ and V to the size (n) of the matrix.

Golub-Reinsch

$$T_{GR}^{(1)} = \frac{70}{3}n^3 + 90\,n^2 + \frac{56}{3}n - 184 \qquad (10.1.1)$$

Hestenes

$$T_{Hes}^{(1)} = [2.1\log_{10}(n) + 2.0]\,\frac{n}{2}(n-1)(14n+59) \qquad (10.1.2)$$

Jacobi

$$T_{Jcb}^{(1)} = [3.1\log_{10}(n) + 1.5]\,\frac{n}{2}(n-1)(24n+104) \qquad (10.1.3)$$

From these equations we see that asymptotically the total operations required by the Golub-Reinsch algorithm will be lower since it is O(n^3) while the other two are O(n^3logn). In fact Figure 10.1.1 shows the Golub-Reinsch algorithm to be superior for all values of n. The chart and the equations also show the Hestenes algorithm to be faster than the Jacobi algorithm. The difference is approximately a factor of two for large values of n ($\approx$ 19n^3logn for Hestenes versus 37n^3logn for Jacobi). This is because the Jacobi algorithm requires the computation and application of two Givens rotations for each step while Hestenes uses one.

Figure 10.1.2 shows the same type of curves for each of the five SVD architectures. The Moreno design implements the Hestenes algorithm exactly and the BLV array implements the Jacobi algorithm exactly. So

$$T_{Mor}^{(1)} = T_{Hes}^{(1)} \quad \text{and} \quad T_{BLV}^{(1)} = T_{Jcb}^{(1)} \qquad (10.1.4)$$

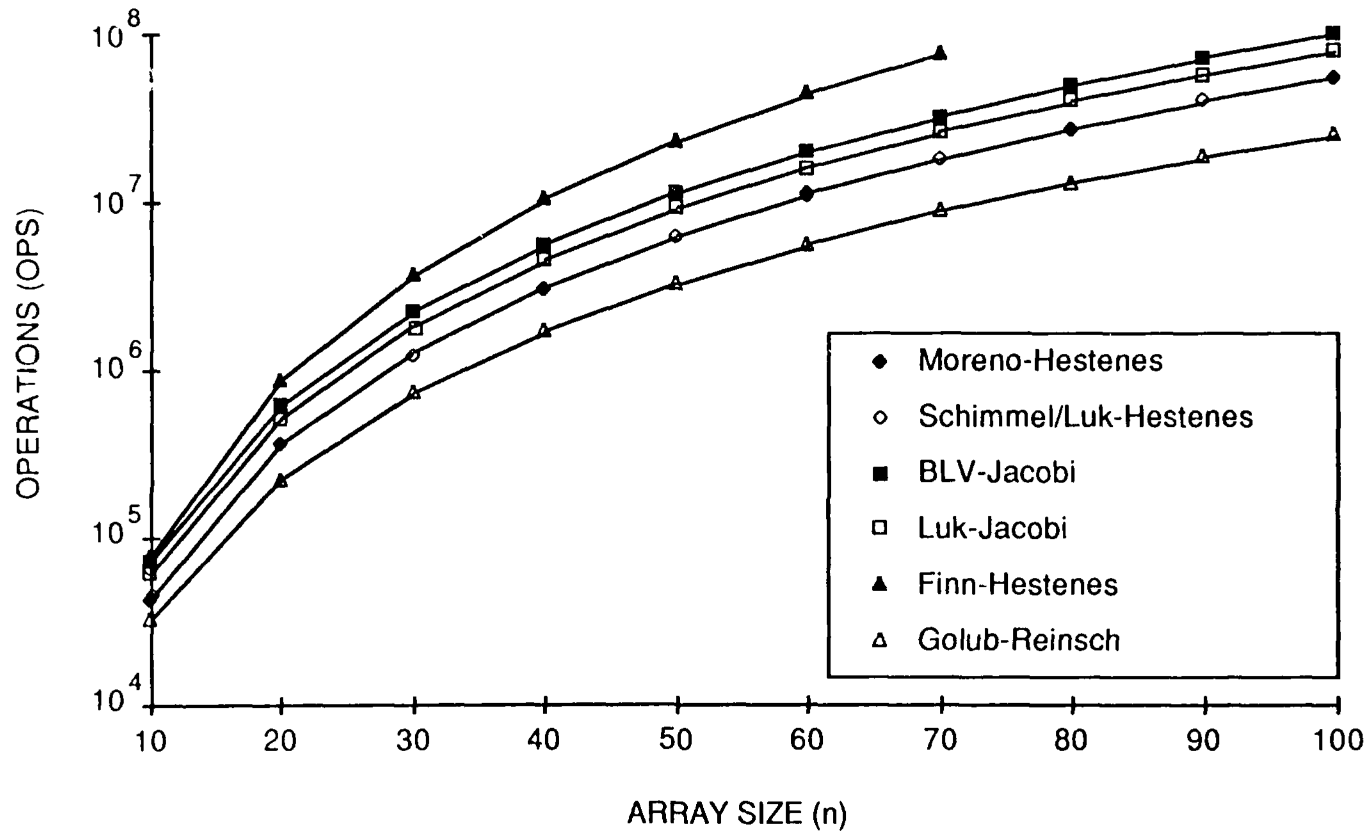


Figure 10.1.2: Total operations required to compute U, Σ and V for the SVD algorithm implemented in each architecture.

The Schimmel/Luk architecture requires slightly more operations than Moreno's since it uses the odd-even rotation ordering scheme. The total number of operations for the Schimmel/Luk algorithm is

$$T^{(1)}_{Sch/Luk} = [2.7\log_{10}(n) + 2.0]\,\frac{n}{2}\,n\,(14n + 29) \qquad (10.1.5)$$

The Luk array uses the Jacobi algorithm on an upper triangular matrix. This yields a small savings in the number of operations in computing the rotations and applying them. The exact formula for the Luk design is

$$T^{(1)}_{Luk} = [3.1\log_{10}(n) + 2.0]\,\frac{n}{2}\,(n - 1)(18n + 84) \qquad (10.1.6)$$

Finally the total number of operations required by Finn's approximate Hestenes algorithm (Method C) is

$$T^{(1)}_{Finn} = (1.95\,n^{0.64})\,\frac{n}{2}\,(n - 1)(14n + 64) \qquad (10.1.7)$$

These relationships are summarized in Figure 10.1.2. The chart shows that the four architectures which implement exact algorithms are similar in the total operations required. Both of the Hestenes based designs are faster than the Jacobi architectures. Finn's algorithm is clearly more expensive. The chart also shows that the best algorithm is more than three times as expensive as the Golub-Reinsch algorithm.

10.2 SVD Computation Times

Figures 10.2.1 and 10.2.2 show plots of the computation time for each architecture. That is these charts show the "wall clock time" (expressed in OPs) needed by the multi-processor units to compute U, Σ and V.

The first chart (10.2.1) gives the results for n up to 1000. As expected, for large values of n the "quadratic" BLV and Luk arrays are much faster with the Luk array providing the best speed. However that the Moreno "linear" design provides better performance than the BLV array for matrices up to n = 200! The Moreno computation time is less than two times that of Luk's for this size matrix. This observation is surprising since we will see that the Luk array has more than 3.5 times as many AUs as the Moreno design for n = 200. The chart also shows that the Moreno design is faster than the Schimmel/Luk design for large values of n. We will see that this is because the Moreno design has more AUs. Finally the chart shows that the Finn architecture is clearly slower than the other quadratic arrays and only "beats" the linear arrays for very large values of n. (The Finn curve will eventually intersect the Moreno curve.)

Figure 10.2.2 shows a detailed plot for $n \leq 100$. This plot is interesting because it shows significant divergences from the long term trends of Figure 10.2.1. For example we see that the fastest architecture for all matrices up to n = 40 is the Schimmel/Luk, linear design. Beyond that point the Luk quadratic array becomes superior. We also see that the Moreno architecture is slower than the Schimmel/Luk design for small matrices (up to $n \approx 55$). This is because the number of AUs in the Moreno design falls off rapidly as n decreases to avoid gaps in the data flow through the pipeline.

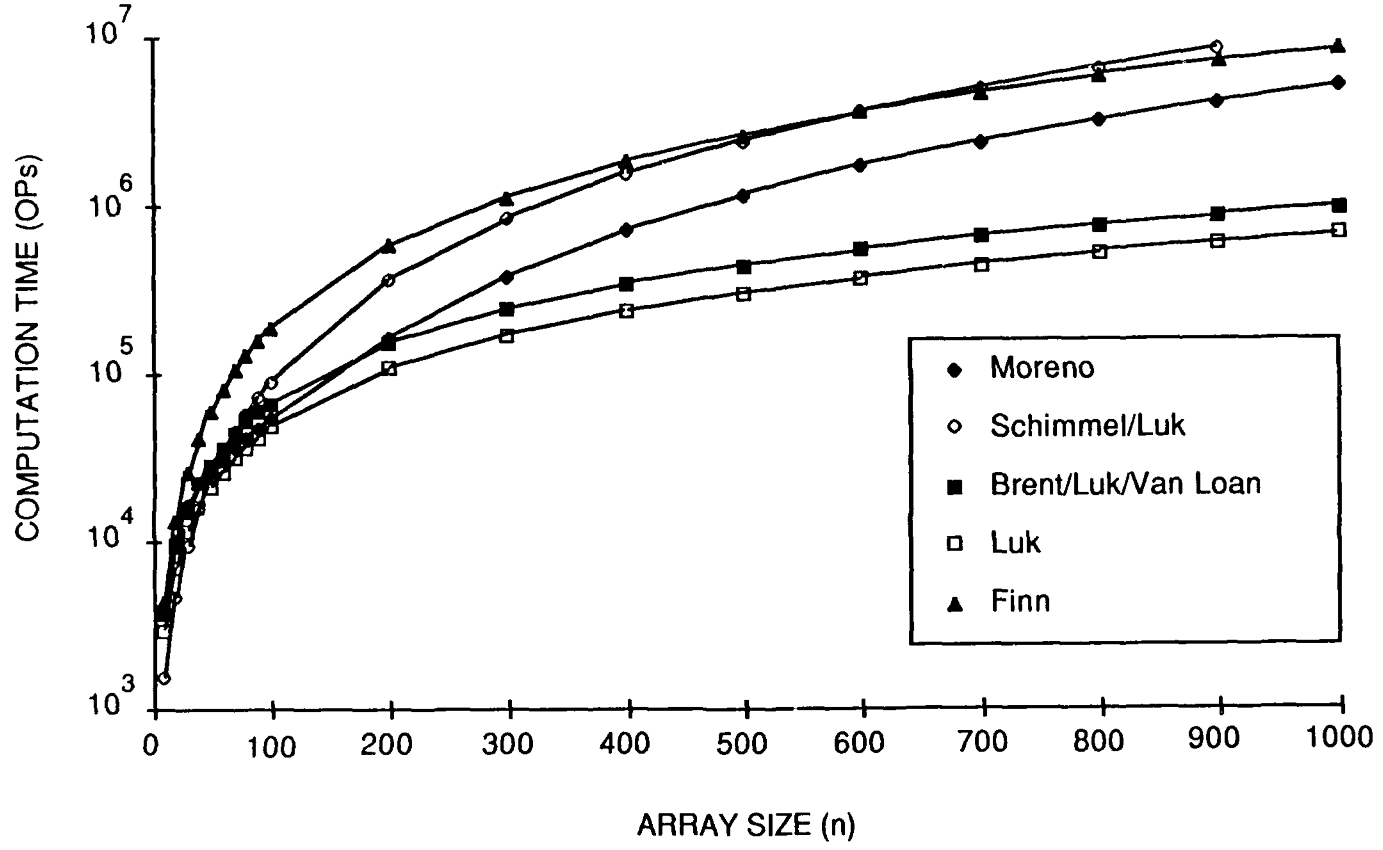


Figure 10.2.1: Computation time for U, Σ and V for different SVD architectures

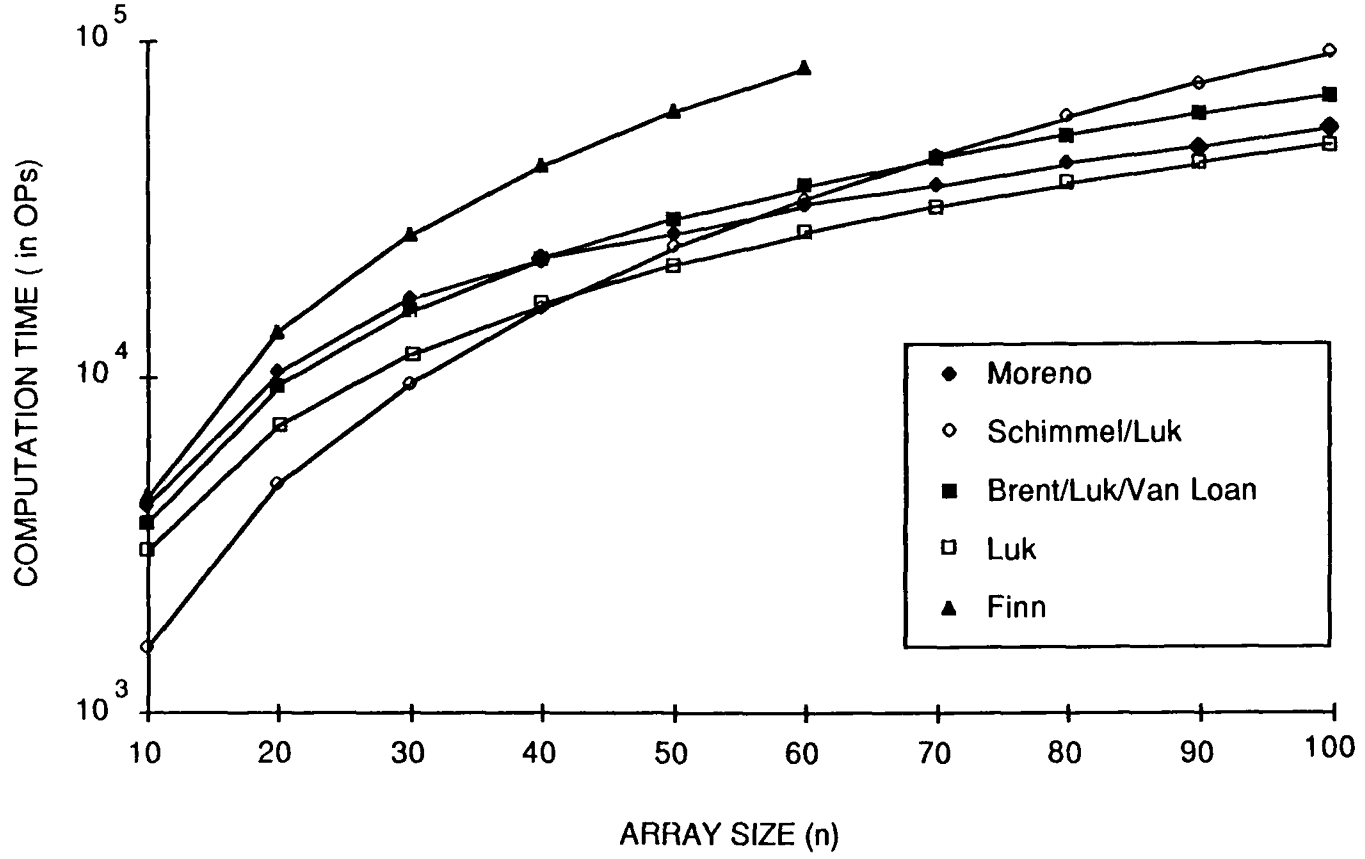


Figure 10.2.2: Computation time for U, Σ and V for different SVD architectures (for small matrices)

10.3 Number of Arithmetic Units

Figures 10.3.1 and 10.3.2 show plots of the number of AUs used by each design. Figure 10.3.1 shows exactly what is expected for large values of n; the linear arrays require many fewer processors than the quadratic arrays. It also shows that the Moreno design has approximately twice as many AUs as the Schimmel/Luk architecture. If we concentrate on the quadratic arrays, we see that the Luk and BLV array require the same number of AUs and the Finn array requires four times as many. This is because the Finn array has twice the number of processors each with twice as many AUs. The other interesting thing to note about this chart is the shear scale of the numbers. For $n = 1000$, the smallest quadratic structure (BLV or Luk) requires approximately 1/4 million AUs! The smallest linear structure (Schimmel/Luk) requires $\approx$ 8000 AUs. These large values make it unlikely that an SVD array for $n = 1000$ will be fabricated in the near future.

Figure 10.3.2 shows the number of AUs required for matrices with more reasonable numbers of elements. In this case there are almost no divergences from the long term trends. The only exception is the Schimmel/Luk architecture. We see that the number of AUs for this design does not drop off as rapidly as for the others. There are two reasons for this. First, the pipelined rotation solver in the Schimmel/Luk design is a "fixed cost" since it has a constant number of AUs (29) for all size matrices. Second the number of AUs in the Schimmel/Luk design is a linear function of n ($=7n+29$). The quadratic arrays have $O(n^2/4)$ AUs. Therefore once n is below 32 the quadratic functions fall off faster than the linear function. Finally, the chart shows that the number of AUs in the Moreno array drops off very quickly as n approaches 10.

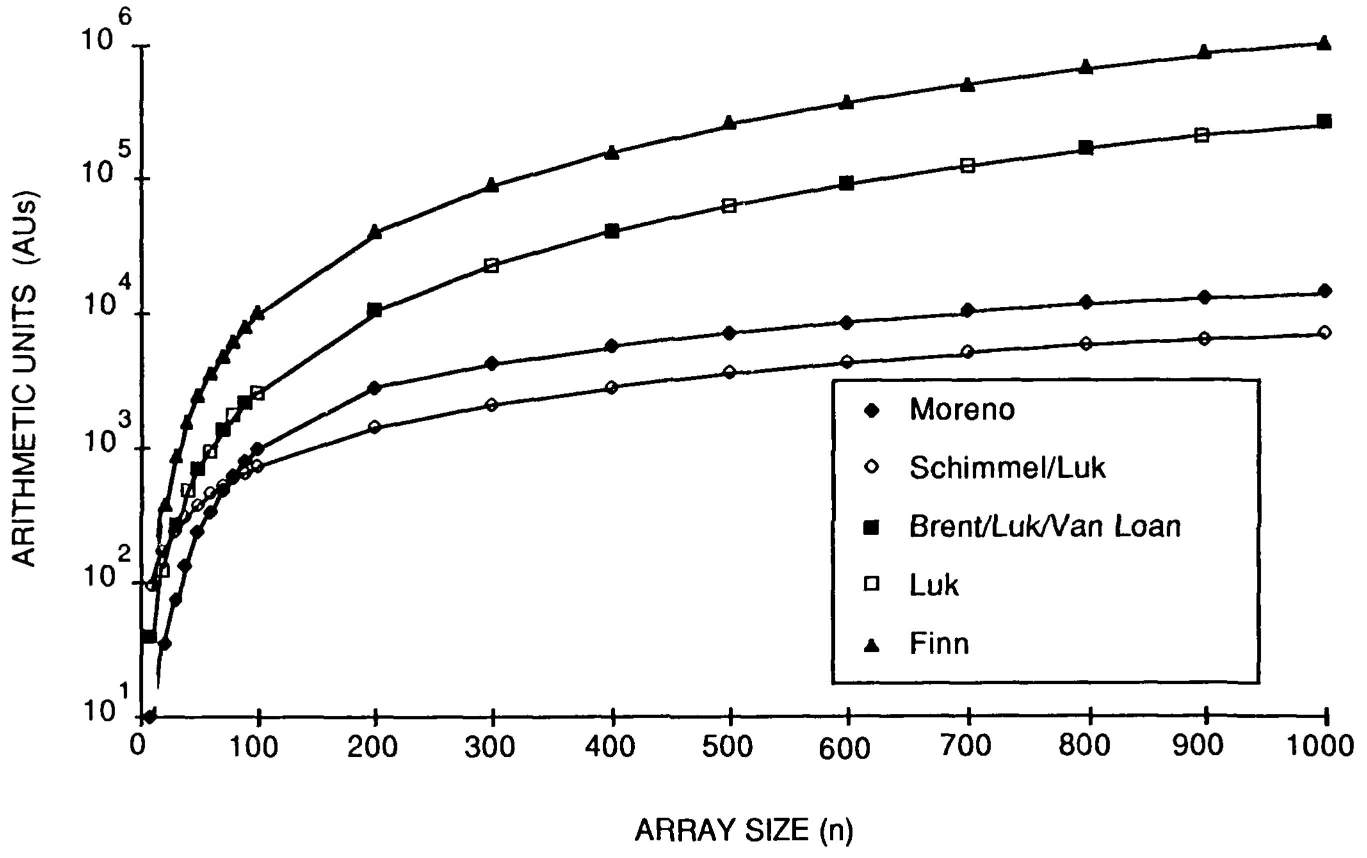


Figure 10.3.1: Arithmetic units required to compute U, Σ and V for different SVD architectures

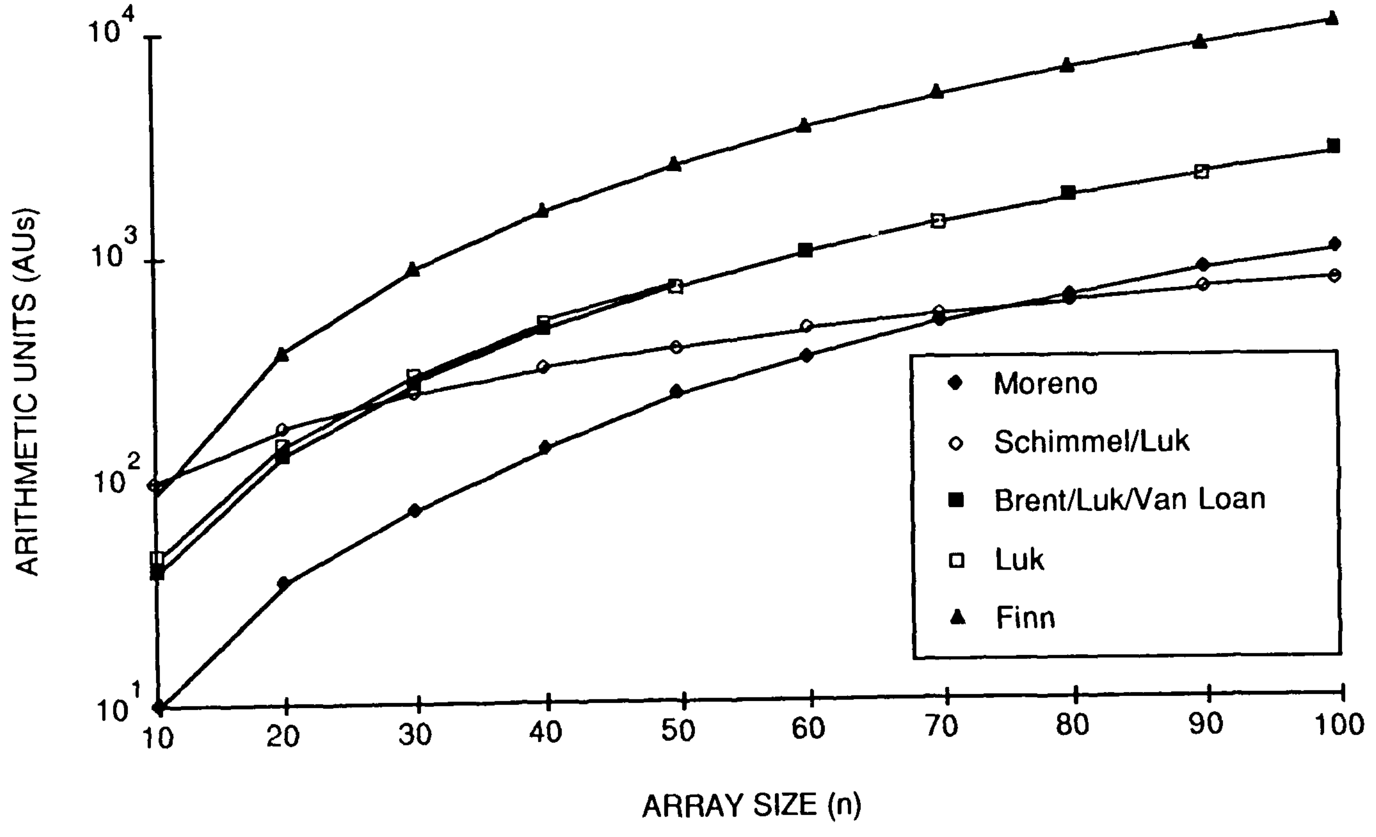


Figure 10.3.2: Arithmetic units required to compute U, Σ and V for different SVD architectures (for small matrices)

10.4 Total Resource Requirements

To this point we have seen that asymptotically the quadratic arrays are faster than the linear arrays but require many more processors. We have also seen that for small matrices ($n \leq 40$) the Schimmel/Luk architecture is the fastest but its hardware requirements are relatively high. In order to give a definitive comparison what is needed is a consolidated figure of merit for each architecture which combines computation time and hardware requirements. Figures 10.4.1 and 10.4.2 attempt to do this by giving plots of the total number of operations that are actually consumed by each architecture during the SVD computation. For example, if an architecture has 10 AUs and takes 1000 OPs to complete the SVD then the architecture consumes 10,000 OPs.

Figure 10.4.1 shows the total resource requirements for large n. It shows that the two linear arrays are very similar and are both superior to the quadratic arrays. To help clarify the differences the following table gives an approximate expression for the the total resource requirements of each architecture for large n and also gives exact values for n = 1000.

Architecture	OPs x AUs	Exact Total (n =1000)
Moreno	$.9n^3\log n$	7.1×10^{10}
Schimmel/Luk	$19n^3\log n$	7.3×10^{10}
Luk	$46n^3\log n$	1.7×10^{11}
Brent/Luk/Van Loan	$69n^3\log n$	2.4×10^{11}
Finn	$99n^{3.64}$	8.3×10^{12}
Golub-Reinsch	$23n^3$	2.3×10^{10}

The table shows that the best quadratic architecture (Luk) is more than 2.5 times as expensive as the linear arrays. The BLV design requires almost three times the amount of resources as the linear arrays. Finn's architecture is clearly inferior

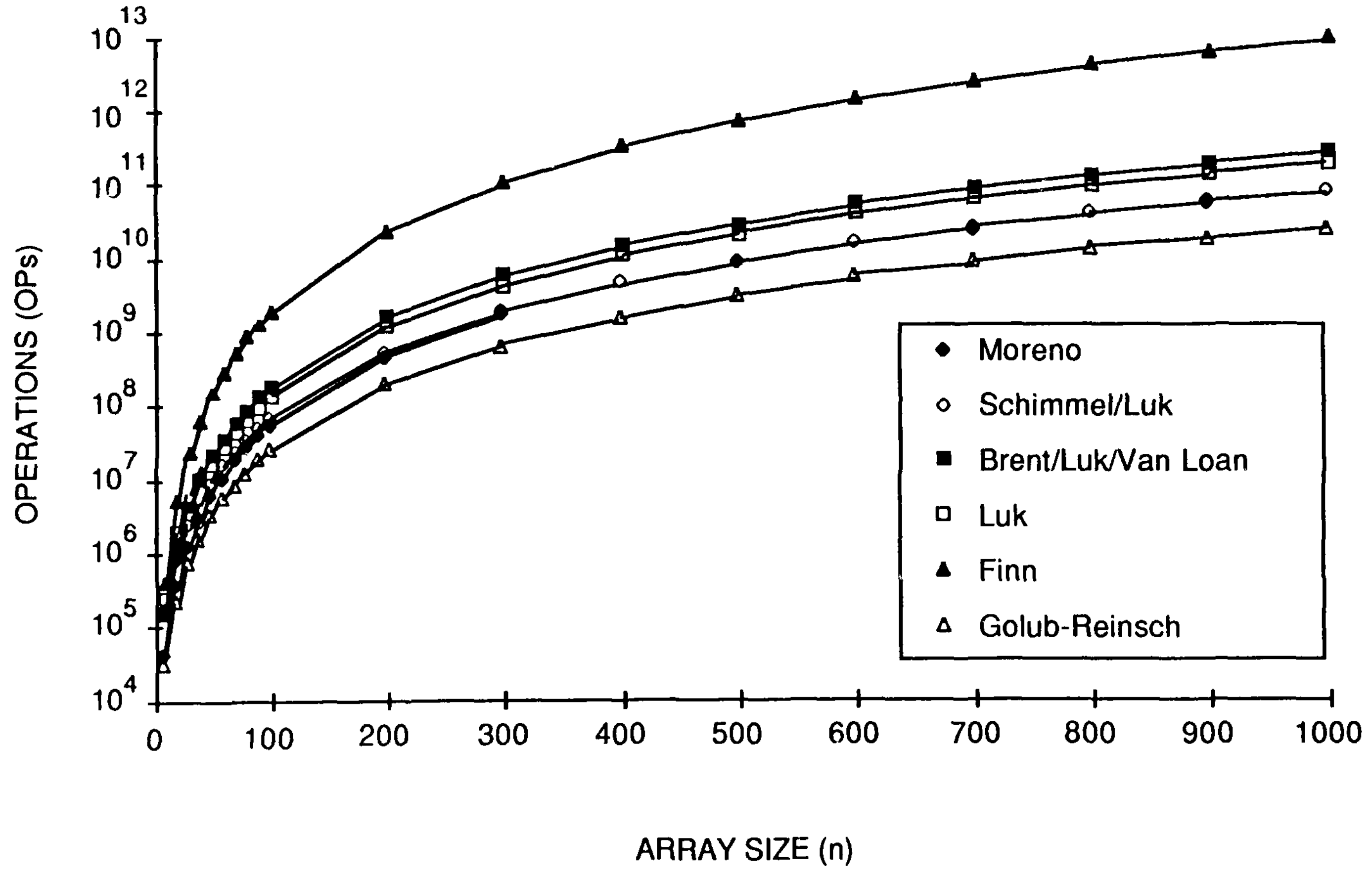


Figure 10.4.1: Total resource requirements (OPs x AUs) of different SVD architectures for the computation of U, Σ and V

since it requires more than an order of magnitude more resources than the next most expensive design. Finally, we see that even the best architecture is three times as expensive as the Golub-Reinsch algorithm.

Figure 10.4.2 shows the total resource requirements for small matrices. Essentially the same conclusions hold for small matrices as for large. The quadratic arrays all continue to be more expensive than the linear arrays.

10.5 Efficiency of the Architectures

To get some insight into the reasons for the conclusions of section 10.4, it is instructive to compute the efficiency of each architecture as a function of n. This can be done very easily by dividing the total computations required by the algorithm implemented by each architecture (given in section 10.1) by the total resources actually consumed (given in section 10.4). That is the efficiency of design x (E_x) is given by

$$E_x = 100 \times \frac{T_x^{(1)}}{T_x C_x} \tag{10.5.1}$$

Using the formulas given in section 10.1 and 10.4, it is easy to develop asymptotic efficiency values for each architecture for large n. The following table gives these figures.

Architecture	Efficiency
Moreno	100%
Schimmel/Luk	100%
Luk	61%
Brent/Luk/Van Loan	54%
Finn	14%

Figure 10.5.1 gives efficiency curves as a function of n for all five architectures. It can be seen that the curves for all designs approach their

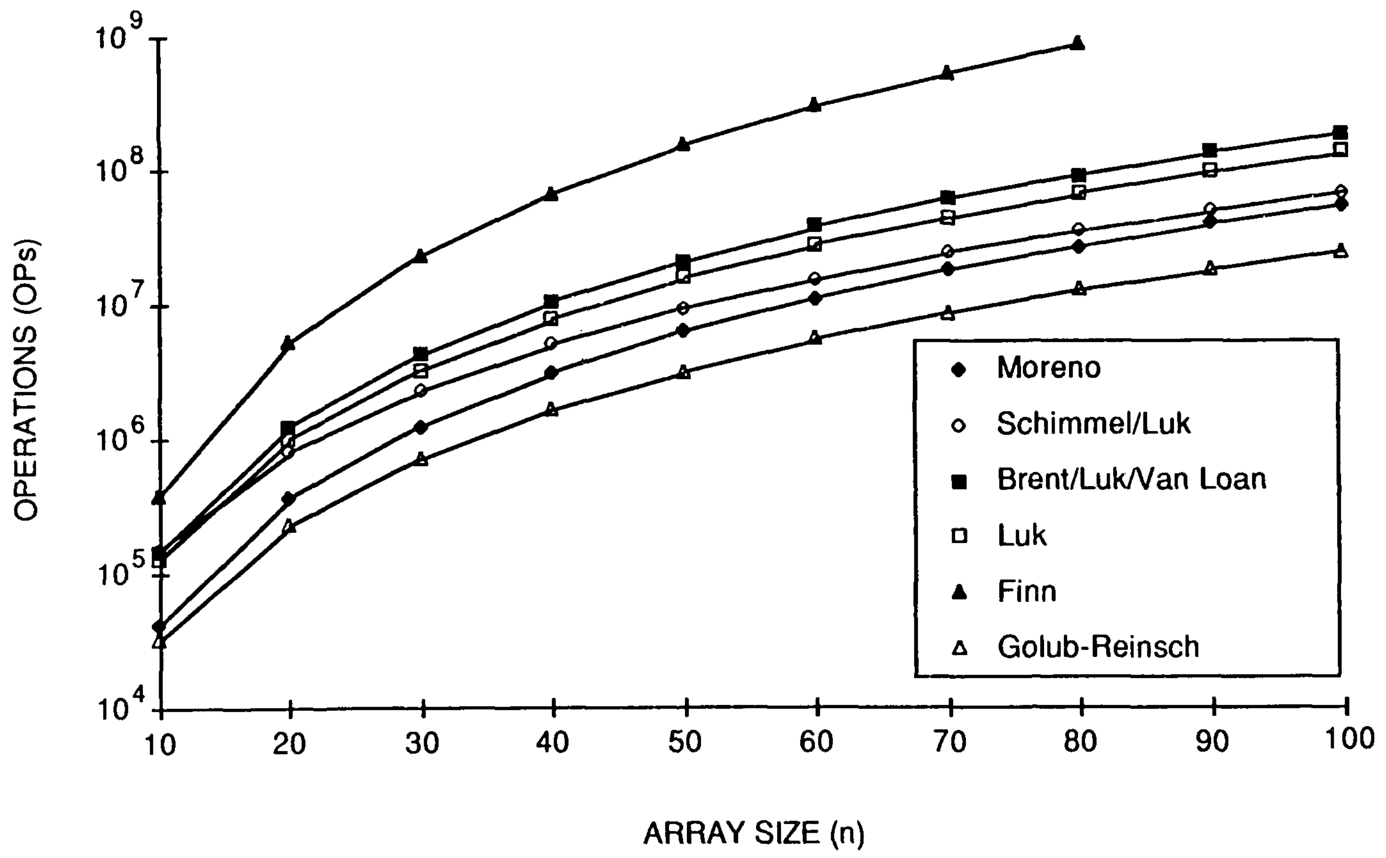


Figure 10.4.2: Total resource requirements (OPs x AUs) of different SVD architectures for the computation of U, Σ and V (for small matrices)

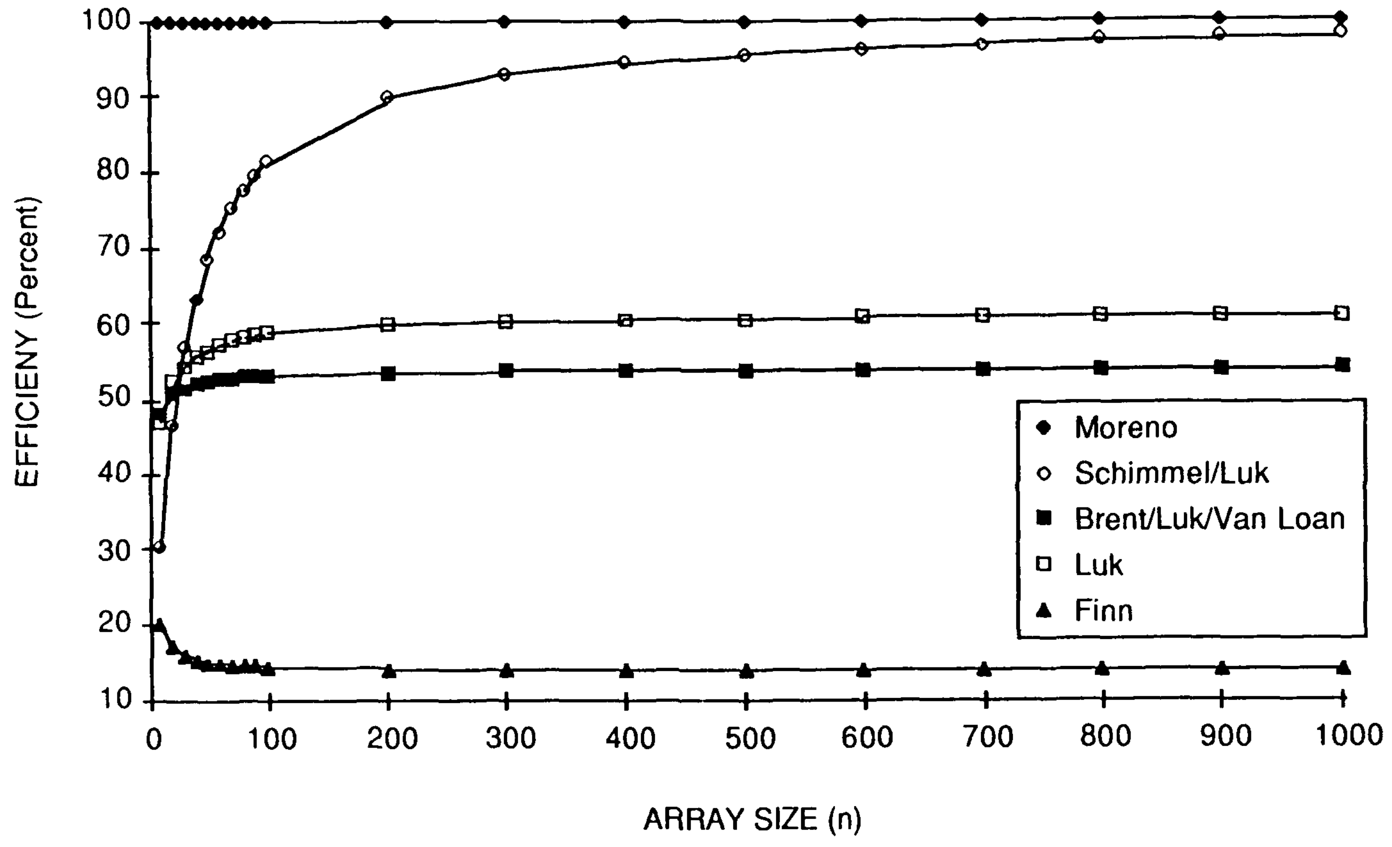


Figure 10.5.1: Efficiency of different SVD architectures for the computation of U, Σ and V.

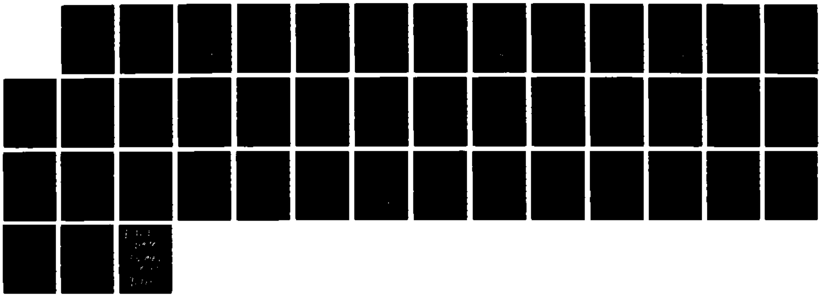
asymptotic values very quickly. The chart also provides very good support for the conclusions drawn in section 10.4. There we said that the linear arrays were less expensive in terms of total resources required. Here we see the reason is that the linear arrays use their OPs very efficiently. In fact when the proper number of AUs are assigned to each stage in the Moreno pipeline essentially full utilization of every OP can be achieved. We also saw in section 10.4 that the performance advantage of the Schimmel/Luk design over the quadratic arrays falls off as n approaches 10. Figure 10.5.1 shows that the reason for this is the sharp decline in the design's efficiency as the data matrix grows smaller. This is just a manifestation of the delay through the pipelined rotation solver. The matrix multiplier and inner product units with their many AUs must sit idle for 20 time steps out of every cycle. For small matrices this delay becomes a significant fraction of the total cycle time. Finally we saw in section 10.4 that the Luk array required fewer OPs than the BLV design and many fewer than Finn's. The chart shows the Luk array is slightly more efficient than the BLV array and four times as efficient as Finn's. Note that if we had used a purely systolic design for the Luk and BLV arrays we would expect efficiencies of 50% and 33% respectively. By using a data flow design we have improved the efficiency of both arrays.

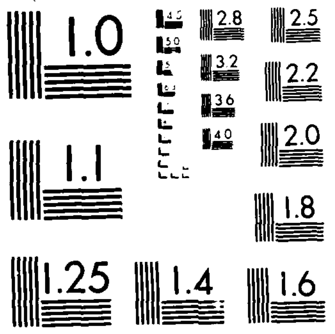
10.6 Speedup Provided by the Architectures

In general the driving force behind the development of parallel architectures is the needed for computation rates which cannot be attained with a serial processor. In the previous sections we have presented much information on the speed of the different SVD architectures but none of it tells us how well we are doing in relation to the best performance of a single processor system. One way to get at such information is to compute the "speedup" provided by each

AD-A196 180 FINITE PRECISION ARITHMETIC IN SINGULAR VALUE DECOMPOSITION ARCHITECTURES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH R A DURYEA AUG 87 3/3

UNCLASSIFIED AFIT/CI/NR-88-119 F/G 12/6 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

architecture. Speedup is defined to be the ratio of the computation time for a single processor system to the time for a multiprocessor system. In our case the best performance for a single processor system is achieved with the Golub-Reinsch algorithm. Therefore the speedup provided by architecture x (S_x) is given by

$$S_x = T_{GR}^{(1)} / T_x \qquad (10.6.1)$$

This function has been computed and plotted for each architecture in Figure 10.6.1. The chart shows that the parallel architectures due offer dramatic speedups for the computation of the SVD. For example, if we use about 3000 AUs in either of the linear designs we can speed up the SVD computation time by approximately a factor of 1000.

Ideally the speedup will be a linear function of the number of AUs [S_x = O(C)].The chart appears to show linear relationships for the speedup of the SVD architectures. In reality the curves all have an O(C/logn) relationship except Finn's which is O(C/$n^{0.64}$). Since C is a rapidly growing function of n, the charts do not show the impact of the logn or $n^{0.64}$ term.

The other desirable feature for a speedup curve is to have a slope close to one. A slope of one indicates that all of the available OPs provided by an architecture are being used productively. Figure 10.6.1 reveals that none of the architectures have a slope close to one. The approximate slope values are given in the following table.

Architecture	Speedup Slope
Moreno	0.37
Schimmel/Luk	0.33
Luk	0.18
Brent/Luk/Van Loan	0.13
Finn	0.02

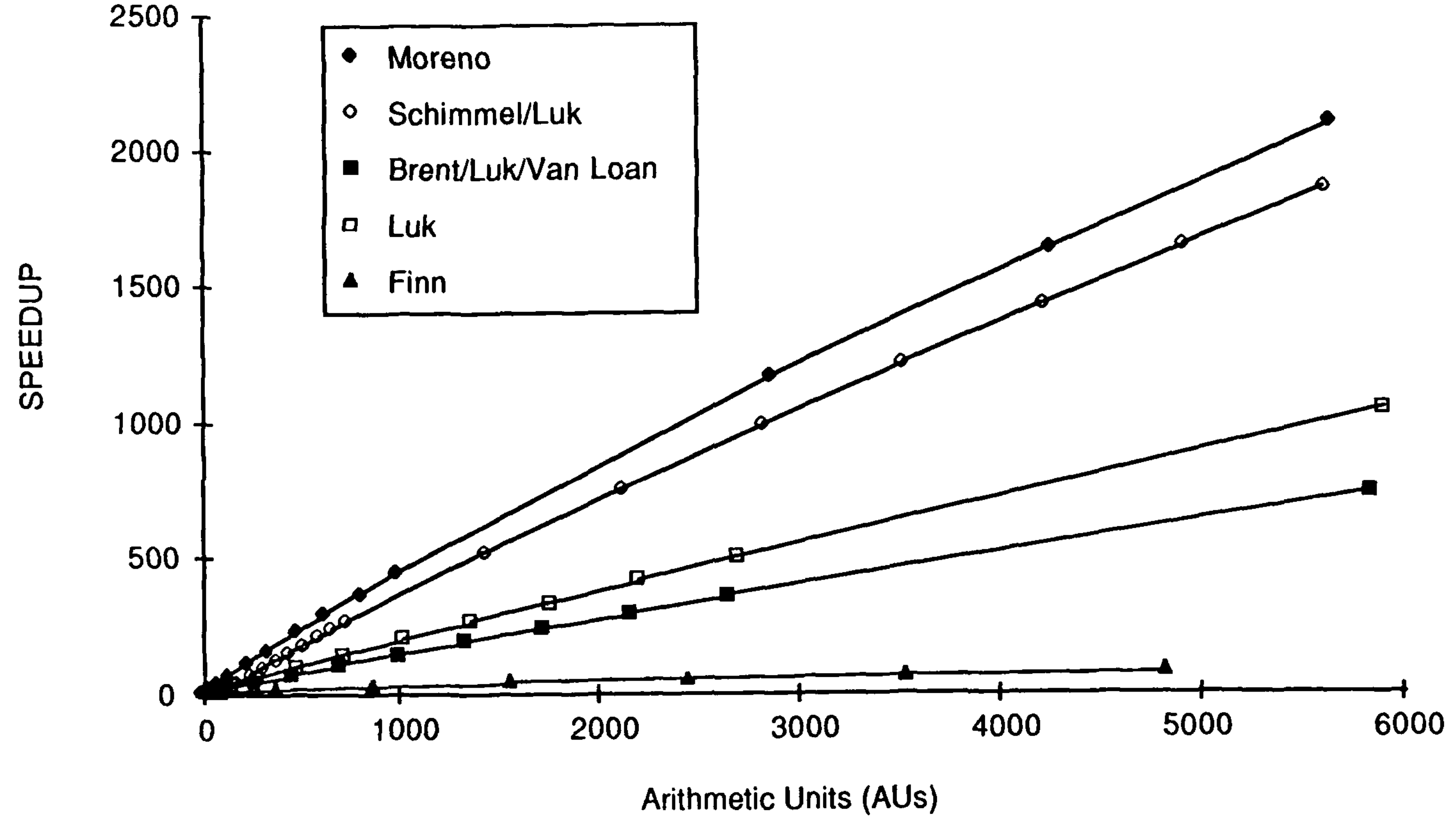


Figure 10.6.1: Speedup provided by different SVD architectures for the computation of U, Σ and V.

This table shows that none of the the SVD architectures is outstanding at applying its OPs towards the task of computing the SVD. In particular the quadratic arrays are very poor. There are two reasons for this. First, as shown in Section 10.1, the Hestenes and Jacobi algorithms require more than twice as many OPs as the Golub-Reinsch algorithm. Therefore in the speedup computation, the SVD architectures start off at a disadvantage. On top of that, the quadratic arrays are all less than 62% efficient.

10.7 Comparison for the Computation of Σ Alone

The comparisons given above are for the computation of U, Σ and V. In many applications only the singular values are required. We can easily perform a similar analysis for the computation of Σ alone. Table 10.7.1 summarizes the characteristics of all of the architectures for the computation of Σ alone.

Comparing this table to Table 10.1 we see that in three cases (Moreno, Schimmel/Luk and Luk) the number of AUs required declines significantly. This is because we can eliminate processors whose sole purpose is to compute V or U. The BLV and Finn arrays require the same number of AUs. The computation times for four of the designs do not change at all. (Finn's decreases by a very small constant factor.) This is because the computation of U and V is done in parallel with Σ in the linear arrays and is performed with available, unused OPs in the quadratic arrays. Therefore elimination of U and V does not effect the computation time for Σ.

Figure 10.7.1 shows a plot of the number of AUs for each architecture. The only significant change from Figure 10.3.2 is that the Luk curves falls below the BLV curve for large n. This is because we have eliminated the extra $n^2/4$ off-diagonal processors required to compute U and V in the Luk array. This leaves

Table 10.7.1
COMPARISON OF SVD ARCHITECTURES
Computing Σ only for an n-by-n matrix

	MORENO	SCHIMMEL LUK	BRENT, LUK VAN LOAN	LUK	FINN
STRUCTURE	Pipelined	Linear Array	Square Array	Triangular Array	Triangular Array
ALGORITHM	Hestenes	Hestenes	Jacobi	Jacobi	Approximate Hestenes
ROTATION ORDER	*Round-robin*	*Odd-even*	*Round-robin*	*Odd-even*	Sequential
SWEEPS/SVD	2.7log(n)+2	2.7log(n)+2	3.1log(n)+1.5	3.1log(n)+2	$1.95n^{0.64}$
OPS/SWEEP	n(n-1)/2, $n \geq 142$ 29.5 n(n-1)/S_θ, n<142	n(n + 23)	89(n-1)	59n	48n
SVD TIME (OPs)	$O(n^2 \log n)$ $n \geq 142$	$O(n^2 \log n)$	O(nlogn)	O(nlogn)	$O(n^{1.64})$
ARITHMETIC UNITS (AUs)	8n+59, $n \geq 142$ $S_\theta(1+8n/59)$, n<142	4n+29	$n^2/4+3n/2$	$n^2/8+2(n-1)$	n(n-1)
OPs x AUs	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^{3.64})$

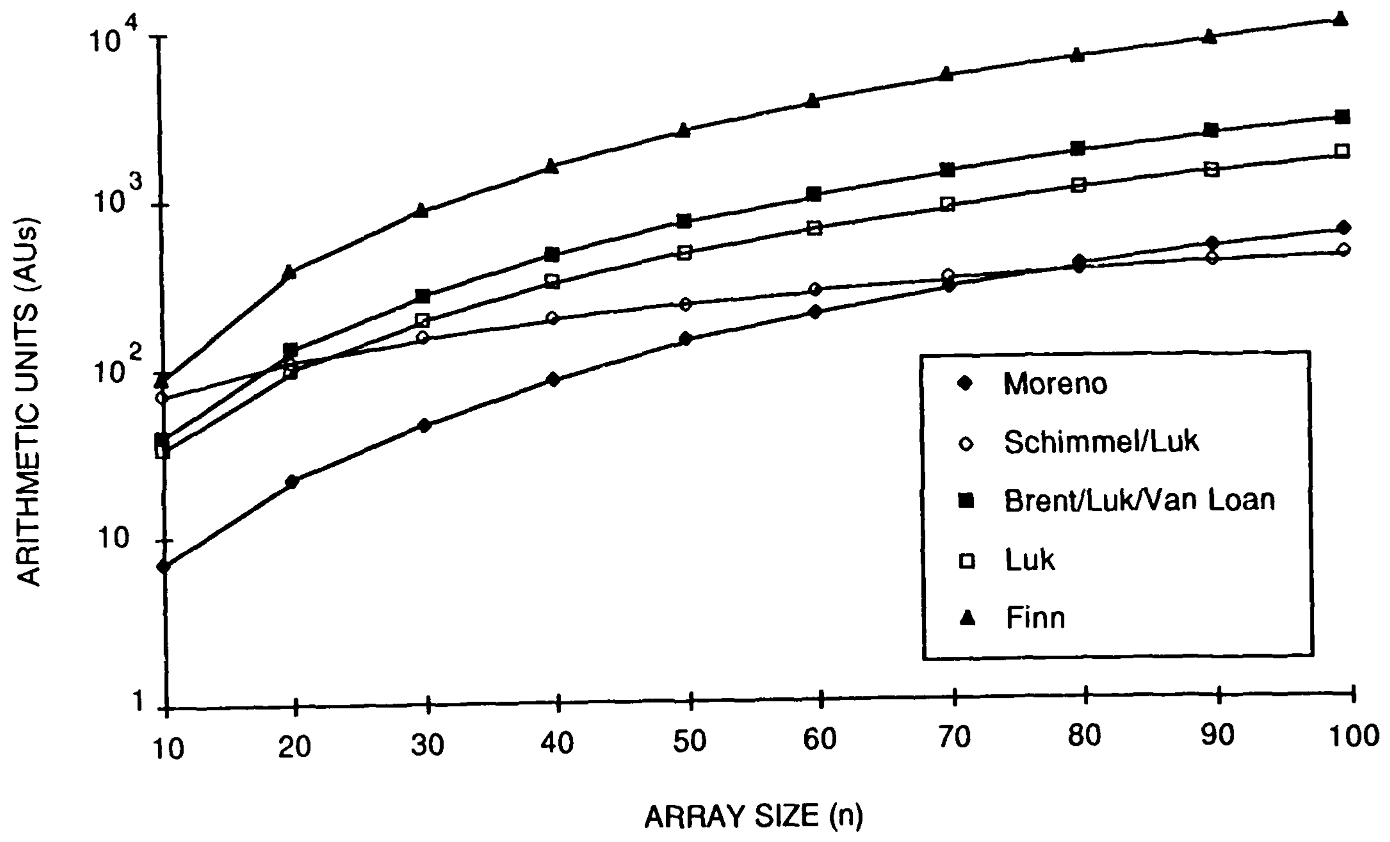


Figure 10.7.1: Arithmetic units required to compute Σ only for different SVD architectures (for small matrices)

$n^2/8$ AUs in the off-diagonal cells of the Luk array, while the BLV array has $n^2/4$.

Figures 10.7.2 and 10.7.3 give the total resource requirements of the architectures for the computation of Σ. We see that all of the conclusions drawn from Figures 10.4.1 and 10.4.2 hold. Note in these two charts that the Golub-Reinsch algorithm requires an order of magnitude fewer total operations than the best architecture. We will see that this is because the Golub-Reinsch algorithm is very efficient at computing Σ.

Figure 10.7.4 gives the efficiency curves for the architectures in computing Σ. Note that the linear arrays maintain the efficiencies displayed in Figure 10.5.1, but the efficiency of the quadratic arrays decreases. This is because we are no longer using their excess capacity which was used to compute U and V.

Finally, Figure 10.7.5 gives the speedup provided by the architectures for the computation of Σ. Here we see that the already bad situation shown in Figure 10.6.1 has gotten much worse. Now the speedup curve for the best architecture (Moreno's) has a slope of only 0.07. The reason for this is that the Golub-Reinsch algorithm is exceptionally efficient for the computation of Σ. If we assume three QR iterations for each singular value, then asymptotically the Golub-Reinsch algorithm requires only $8n^3/3$ OPs to compute Σ. This is only 9% of the total OPs required to compute U, Σ and V. The Hestenes and Jacobi algorithms do not share this property. The Hestenes algorithm uses 57% of its OPs for Σ and Jacobi uses 50% (asymptotically). Therefore in the speedup computation, the parallel architectures are being compared to an exceptionally efficient single processor algorithm. The result is a very poor showing for the parallel architectures.

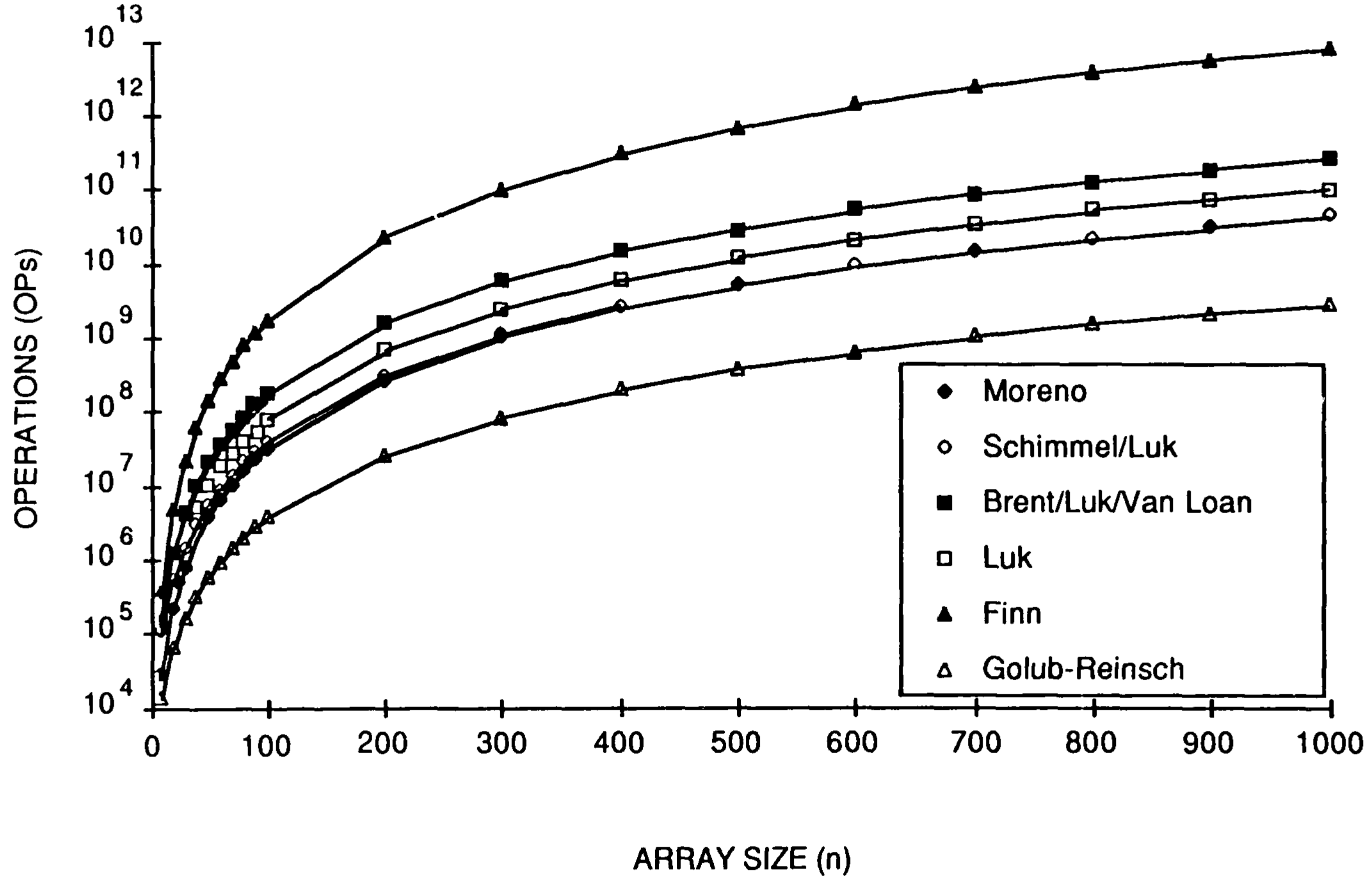


Figure 10.7.2: Total resource requirements (OPs x AUs) of different SVD architectures for the computation of Σ only.

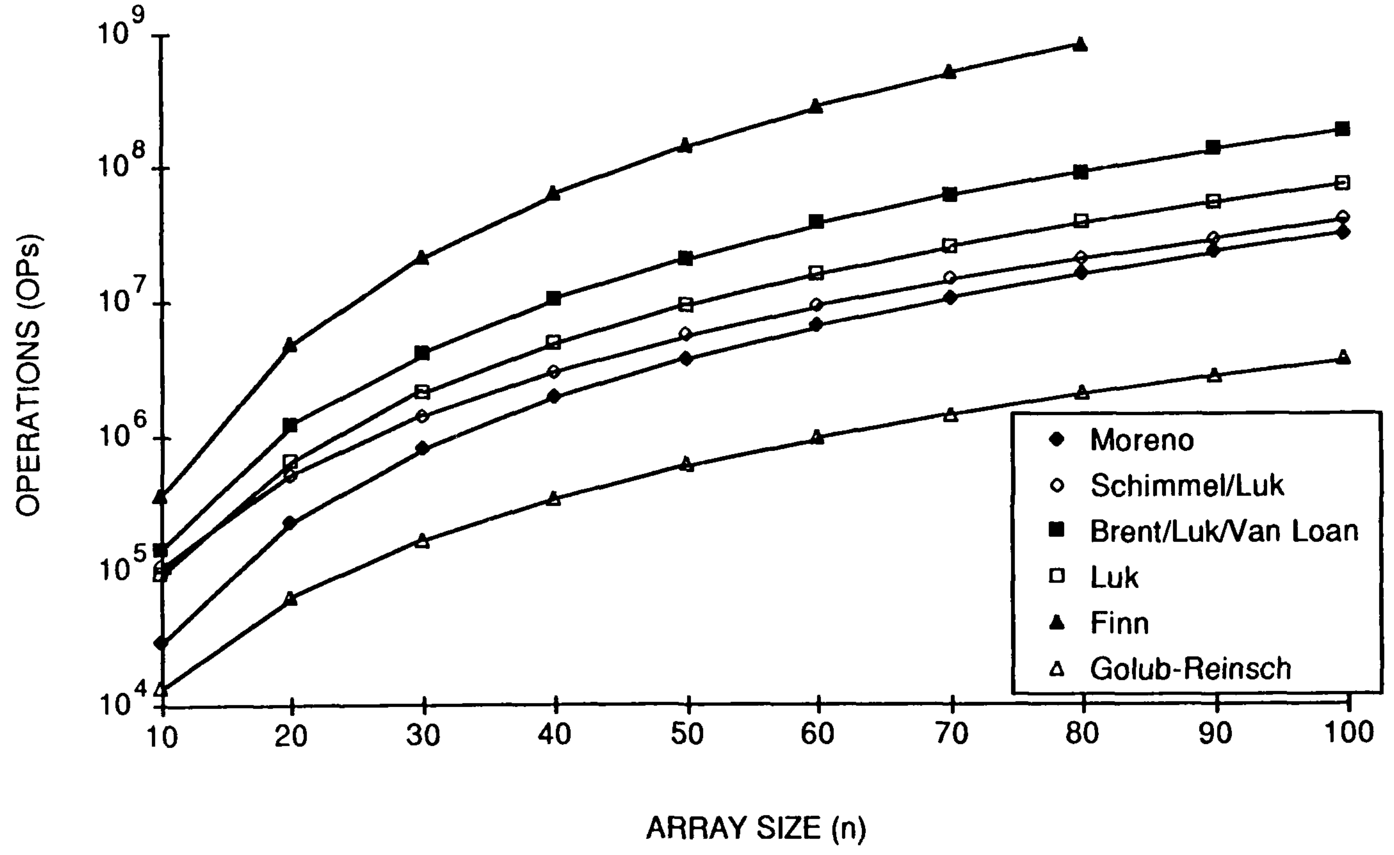


Figure 10.7.3: Total resource requirements (OPs x AUs) of different SVD architectures for the computation of Σ only (for small matrices)

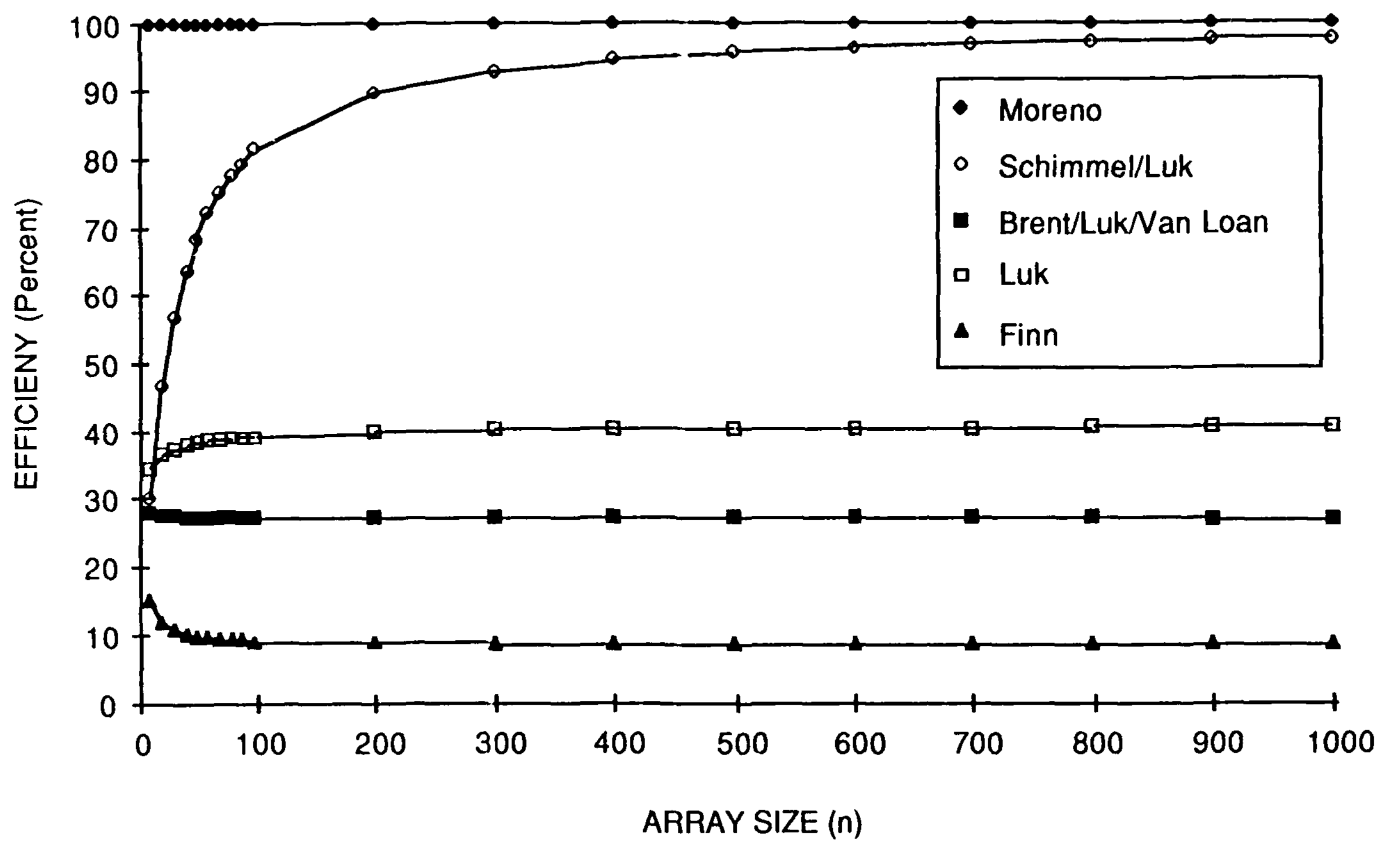


Figure 10.7.4: Efficiency of different SVD architectures for the computation of Σ only.

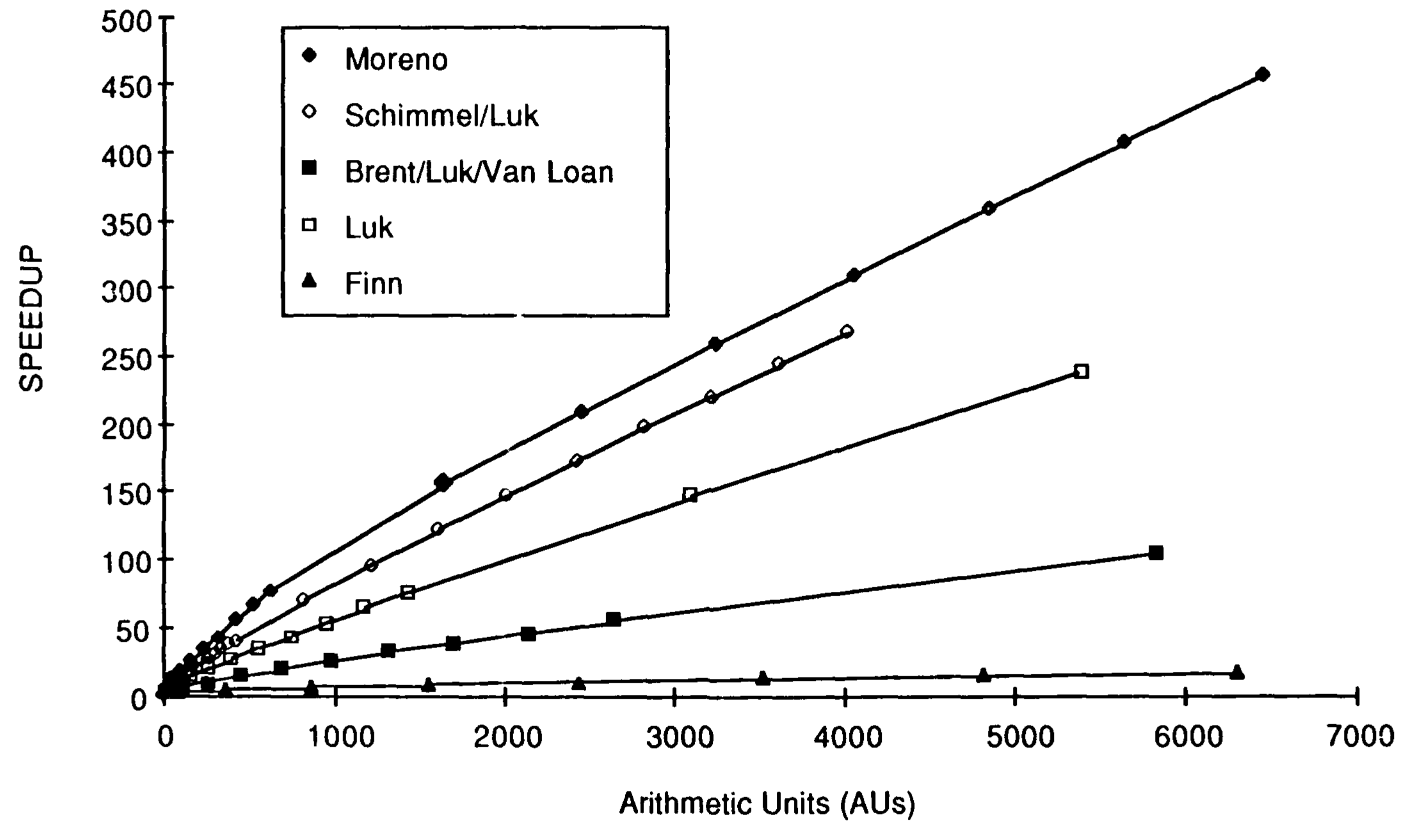


Figure 10.7.5: Speedup provided by different SVD architectures for the computation of Σ only.

10.8 Area Requirements of the Architectures

So far we have looked at the computation time and arithmetic unit requirements of each architecture. When architectures like these are implemented in VLSI a critical concern is the requirement for chip area. The number of AUs is certainly a primary driver for the total area, but there are additional factors which must be considered. Such concerns include the wiring area required by the interconnection pattern, the amount of storage that must be provided, and the complexity of the control structure. We will take a qualitative look at these concerns to determine which architectures are likely to require extra area.

With a few exceptions, all of the architectures have local interconnection wiring patterns. The Moreno design requires broadcasting of the rotation parameters and has global wiring patterns from memory to the pipeline. Otherwise the majority of the Moreno architecture has simple tree-like wiring patterns. The Schimmel/Luk architecture also requires global routing to memory. The Finn architecture has global end around connections. The remaining interconnections in these two architectures are predominantly vertical and horizontal nearest neighbor connections. The BLV and Luk architectures have only local interconnections but they both require diagonal connections. These will significantly increase the wiring area in VLSI or will require the use of additional wiring layers. So the BLV and Luk arrays will require relatively more area for interconnect than the other three designs.

In terms of area required for storage of data values, the linear arrays have several advantages. Both the Moreno and Schimmel/Luk arrays have global memories to store the data matrices. Since the memories are consolidated, they can be designed using many of the area conservation techniques used in the design of random-access memory chips. The linear arrays require only one or two registers per processor since each AU performs only one operation then immediately transmits the data to the next AU. The quadratic arrays will require many registers per cell. In contrast to the linear arrays, they store all of the elements of the U, Σ and V matrices in individual registers in the processor cells. They also perform many operations before transmitting data to the next cells. This necessitates the storage of intermediate results. Finally the iterative algorithms for division and square root require the storage of a table of initial values. For the quadratic arrays this table is duplicated in every diagonal processor, or in the case of Finn's architecture, in every cell. The linear arrays require only one such table. With these advantages, the area devoted to memory elements in the linear designs should be relatively much lower than that of the quadratic arrays.

The same statement can be made about the control structures required by the architectures. In effect the control structure for the linear arrays is built right into the arrangement of the AUs and their interconnection pattern. Each processor repeatedly performs only one operation and then sends its result to the next. The only element of the designs which appears to need a dedicated control structure is the global memory. These statements are always true for the Schimmel/Luk array. They are also true for the Moreno array for values of n above 142 since in this range each stage processes the data for an entire column all at once. Below that value, the Moreno architecture would require some control mechanism at the front end of each major pipeline stage. Such a controller is needed to divide each column into fixed length segments and control the flow of the segments. In contrast the quadratic arrays would require control structures in each cell. Since each cell performs a number of operations using a limited number of AUs, a controller is needed to assign tasks to AUs and to direct the flow of operands. It is likely that each of the cells in the quadratic arrays would require a simple CPU to control the AUs.

In summary, the quadratic arrays require significantly more chip area than the linear arrays. They require many more AUs and much more area per AU.

10.9 Summary and Conclusions

This chapter has compared the different architectures for computing the SVD . Based on this comparison we can draw the following conclusions.

a. The total resource requirements (OPs x AUs) of the linear arrays are lower than the quadratic arrays for all size matrices.

b. The computation time for the linear arrays is lower than that of the quadratic arrays for all square matrices up to n = 40. The fastest time for the linear arrays is less than two times that of the fastest quadratic array (Luk's) for all matrices up to n = 200. The theoretical speed advantage of the quadratic arrays is really apparent only for arrays larger than n = 200.

c. The speed advantage of the quadratic arrays for large matrices is obtained by a dramatic increase in the number of AUs. For example at n = 200 the fastest linear array (Moreno's) requires approximately 2860 AUs. For the same size matrix the fastest quadratic array (Luk's) requires approximately 10400, more than 3.5 times more!

d. Each AU in the quadratic arrays will require more chip area than a similar AU in the linear arrays due to the overhead area that must be devoted to memory, control and interconnect.

e. All of the architectures are driven in some way by the rotation angle computation. Both linear arrays have complex, pipelined, rotation units made up of many AUs. These units are required to maintain a throughput rate high enough to support the rotation application units. The quadratic arrays cannot utilize pipelined rotation solvers. They pay the price of the rotation computation by having a few processors perform many operations. This causes the computation time for the quadratic arrays to be much longer than would otherwise be the case.

f. The linear arrays make very efficient use of the available OPs. The quadratic arrays waste more than half even using a data flow structure.

g. The speedup provided by the parallel SVD architectures over the serial Golub-Reinsch algorithm is potentially very large but is far from optimum. For the computation of U, Σ and V the best architecture (Moreno's) provides an asymptotic speed up that is only 37% of the theoretical maximum. For the computation of Σ alone the speedup provided by the parallel architectures is very low. The best architecture provides only 7% of the theoretical maximum speedup. The Golub-Reinsch algorithm is just far superior to the Hestenes and Jacobi algorithms for the computation of Σ.

h. Overall, the Schimmel/Luk architecture appears to be the best for implementation with current VLSI technology. It has a very simple, regular structure which scales directly with the number of rows in the data matrix. Almost all of its cells are identical multiply-accumulate processors. It has a very simple interconnection pattern. While we have focused exclusively on square matrices, it is important to note that the Schimmel/Luk design requires no modification other than increased memory to handle rectangular arrays. Finally, with current VLSI technology it is difficult to imagine building SVD arrays for n much larger than 40. The Schimmel/Luk array provides the fastest computation time for matrices of this size.

11.0 COMPARISON OF SVD ARCHITECTURES WITH CORDIC AUs

11.1 CORDIC Processors in SVD Architectures

As stated in the conclusions of the previous chapter, all of the SVD architectures have difficulty with the computation of the rotation parameters required by the Hestenes and Jacobi algorithms. This situation arises because it is difficult to compute divisions and square-roots with floating point multipliers and adders. One potential solution to this problem is the use of CORDIC processors to compute and apply rotations. Using CORDIC processors can greatly simplify the structure and data flow in the SVD architectures. As an example Figure 11.1.1 shows the dependency graph for the rotation computation and norm update function in the Hestesnes algorithm using CORDIC units. Comparing this figure to Figure 9.4.1.1, which shows the same computation with floating point AUs, we see a dramatic reduction in the complexity of the calculation. This clearly illustrates the potential benefits of CORDIC processors for the SVD architectures. In this chapter we will analyze the five SVD architectures to determine if they can employ CORDIC units in place of floating point AUs. If so we will compare the CORDIC versions to the floating point versions to show the impact of the CORDIC processors.

11.2 Operation Time and Area Requirements for a CORDIC Arithmetic Unit

In order to perform comparisons between CORDIC and floating point architectures, we must have some idea of the relationship between the time for a CORDIC operation and a floating point OP. We must also determine if there is a significant difference in the area requirements of a CORDIC and floating point AU. This is difficult because there have only been a few CORDIC processors

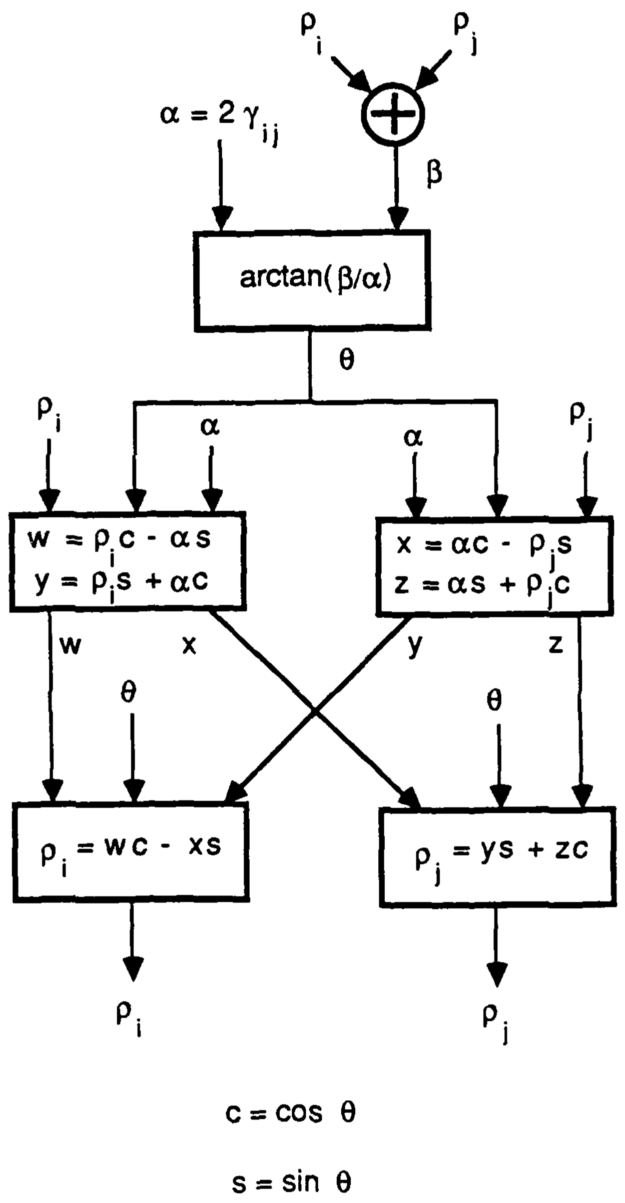


Figure 11.1.1: Dependency graph for the rotation angle computation/norm update (θ/NU) function of the Hestenes algorithm with CORDIC processors.

built and even fewer implemented in VLSI. We will draw some general conclusions based on the limited data that is available.

First, it appears that the VLSI area requirements of a CORDIC unit and a floating point AU are roughly the same. With current CMOS technology either type of unit consumes the majority of a chip. The one well documented VLSI CORDIC processor was developed by Haviland and Tuszynski in 1980 [Hav80]. They constructed a CORDIC processor which computed 24 bit fixed point results. The chip was constructed using an 8-µm, metal-gate, bulk CMOS process. In order to fit the entire algorithm onto a single chip, the 24-bit data were processed in two 12-bit steps. All of the hardware was sized for 12-bit computations. With currently available 2 µm CMOS technology a "full' 24-bit version, or perhaps even a 32 bit version, of the same architecture could be fabricated on a single chip.

In order to make some estimate of the computation time for a CORDIC operation we must make an assumption on the type of arithmetic used. In order to compute an n-bit result in a CORDIC processor a series of n shifts and adds must be performed. Some CORDIC processors have been built which use floating point arithmetic for these operations. However this does not make sense for the SVD calculation. For instance, If floating point arithmetic is used in a CORDIC processor then it would take thirty two floating point OPs to apply a rotation to a pair of 32-bit matrix elements. However if the rotation is applied with floating point AUs it takes only 12 OPs. The real advantage of the CORDIC algorithm comes through the use of fixed point arithmetic like that used in the Haviland and Tuszynski chip. In this case the shifts and adds can be performed very rapidly using fixed point hardware. We will assume that fixed point arithmetic is used in the CORDIC algorithm.

With this assumption we can determine the time required for a CORDIC OP in two ways. First we can extrapolate the data given by Haviland and Tuszynski for their chip to today's CMOS technology. We can also break a CORDIC OP into its individual steps and estimate a time for the steps.

Haviland and Tuszynski estimated that their chip could perform a CORDIC operation in 40 µs [Hav80]. This was based on an estimated gate (inverter) delay of 100 ns for the 8 µm, metal-gate, CMOS process. With current 2 µm CMOS technology, typical gate delays are on the order of 5 ns. Fabrication of the same architecture with 2µm CMOS should produce a chip which could compute CORDIC OPs in 2 µs. Another factor of 2 increase in speed can be achieved by using full 24 bit operands rather than dividing the computations into 12-bit half-words. Therefore the Haviland and Tuszynski data suggest a CORDIC OP time of approximately 1 µs or 1000 ns.

We can also develop a time estimate based on the fact that a single CORDIC OP requires n fixed point additions of n-bit numbers to be computed in a serial manner. For $n = 24$, current technology allows the computation of an n-bit fixed point addition in approximately 30-40 ns. Based on this, the total time for a 24-bit CORDIC OP would be 720-960 ns.

Both of our estimates indicate a CORDIC OP time of approximately 1000 ns. This is 10 times the time for a floating point multiply (100 ns) with current technology. We will use this 10 to 1 ratio as the basis for comparison of CORDIC and floating point SVD architectures.

11.3 CORDIC Versions of the SVD Architectures

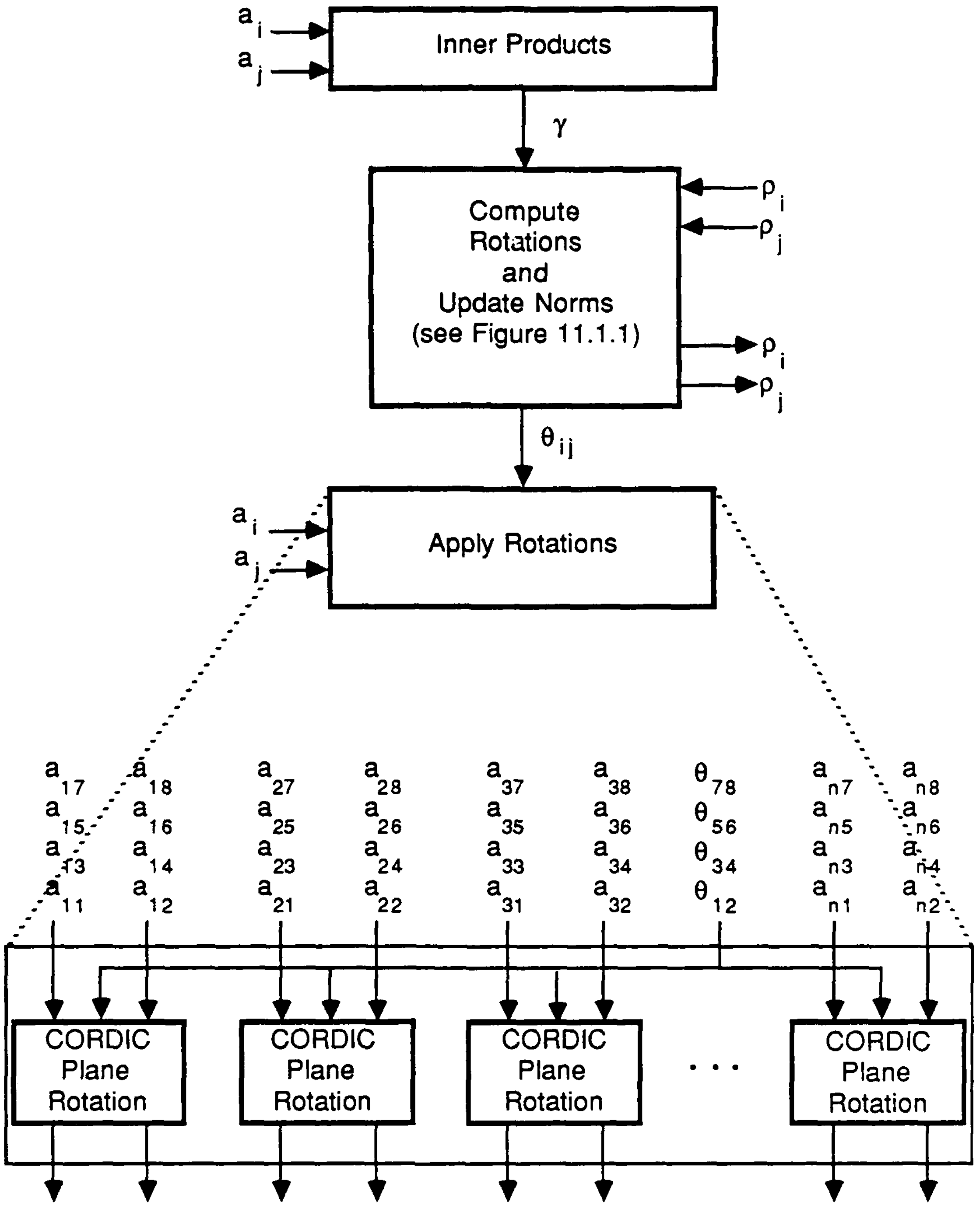
In this section we will analyze each of the five architectures to determine if they can employ CORDIC units in place of floating point AUs. If so we will analyze the number of CORDIC units required by an architecture and the computation time provided by the CORDIC design. Note that we will use the term AU to refer to an arithmetic unit which performs either CORDIC or floating point operations. The term OP will still be limited to mean the time for a floating point OP. When we want to indicate the time for a CORDIC operation we will use the term CORDIC OP.

11.3.1 Moreno Pipelined Architecture

The Moreno architecture can utilize CORDIC processcrs very effectively in the rotation computation and rotation application units shown in Figure 8.1.2. CORDIC versions of these units are shown in Figure 11.3.1.1. Note that in this case the actual rotation angle (θ) is broadcast to the rotation application unit instead of the cosine and sine of the angle. Since the inner product unit performs only multiplications and additions CORDIC processors would not be appropriate. Instead the original structure with floating point AUs will be retained.

Analysis of the computation time and the number of AUs required by the CORDIC design is much easier than was the case for the floating point design. In the floating point version, the Moreno pipeline could have as many as 62 + $\log_2(n)$ stages. This lead to two distinct computations of the number of AUs, one for $n \geq 142$ and one for $n < 142$. For the CORDIC design there are far fewer stages since there are a maximum of three in the θ/NU unit and one in the rotation unit.

Also for the floating point design with $n \geq 142$, we saw that the fastest computation time was achieved by assigning enough AUs to each stage to give a throughput rate of one orthogonalization per time step. The same is true for the CORDIC design. Once the data matrix is sufficiently large, we can achieve a rate of one orthogonaliztion per time step by assigning five CORDIC AUs and 1



a_i
a_j
Inner Products
γ
Compute Rotations and Update Norms (see Figure 11.1.1)
ρ_i
ρ_j
ρ_i
ρ_j
θ_{ij}
a_i
a_j
Apply Rotations
a_{17} a_{15} a_{13} a_{11}
a_{18} a_{16} a_{14} a_{12}
a_{27} a_{25} a_{23} a_{21}
a_{28} a_{26} a_{24} a_{22}
a_{37} a_{35} a_{33} a_{31}
a_{38} a_{36} a_{34} a_{32}
θ_{78} θ_{56} θ_{34} θ_{12}
a_{n7} a_{n5} a_{n3} a_{n1}
a_{n8} a_{n6} a_{n4} a_{n2}
CORDIC Plane Rotation
CORDIC Plane Rotation
CORDIC Plane Rotation
. . .
CORDIC Plane Rotation

Figure 11.3.1.1: CORDIC version of the Moreno architecture

standard AU to the θ/NU unit and 2n CORDIC AUs to the rotation application unit (n AUs each for the H matrix and the V matrix). We also need standard AUs to apply the CORDIC constant and compute inner products. Note that since the CORDIC AUs are 10 times slower than floating point AUs, we can reduce the number of floating point AUs in the inner product ui..t. We can also apply the CORDIC constant to the output of more than one CORDIC AU in the rotation application unit with a single multiplier. To match the throughput rate of the other stages the number of AUs in the inner product unit can be reduced by a factor of 10 from 2n to n/5. Similarly we need only n/5 multipliers to apply the CORDIC constant. Overall the CORDIC version of Moreno's architecture requires 2.4n + 6 AUs. This is approximately 6 times less than the number of AUs in the floating point version of Moreno's architecture for $n \geq 142$ (14n + 59).

This formula for the number of AUs applies when we have enough columns in the data matrix to keep all stages filled. Since there are 4 stages in the rotation units and $\log_2(n/5)$ in the inner product unit, this condition will be satisfied when $n \geq 2 \times [4+\log_2(n/5)]$. This equation is true for all $n \geq 10$. So the formula for the number of AUs in the CORDIC design applies over the full range of interest.

As stated above, with this allocation of AUs the throughput rate of the pipeline is one orthogonalization per time step. Each time step is the length of time for a CORDIC operation or equivalently 10 OPs. Therefore the total time for the SVD computation for the CORDIC version of the Moreno architecture ($T_{Mor(co)}$) is given by

$$T_{Mor(co)} = 10\,[2.7 \log_{10}(n) + 2.0]\,\frac{n}{2}\,(n-1) \text{ OPs} \qquad (11.3.1.1)$$

This is 10 times the computation time of the floating point design with $n \geq 142$.

11.3.2 Schimmel/Luk Architecture

It is not possible to use CORDIC processors efficiently in the Schimmel/Luk architecture. Their design is very carefully tailored to allow the rotations to be applied by a matrix-matrix multiplication. Such an operation requires only multiplications and adds. CORDIC arithmetic units are very inefficient for these operations.

11.3.3 Brent/Luk/Van Loan Array

The BLV architecture is ideally suited to CORDIC units since its cells only compute and apply rotations. Figure 11.3.3.1 shows the dependency graphs for the computations in the diagonal and off-diagonal processors using CORDIC operations. If we compare this figure to Figure 9.5.2.1 we can see that the rotation computation is greatly simplified. Also note that we now transmit the actual rotation angles (θ_1 and θ_2) from the diagonal processors rather than the corresponding cosines and sines.

Figure 11.3.3.1 shows that we can use two CORDIC AUs in the diagonal processors and either one or two in the off-diagonal processors. We also need one floating point AU in each cell to apply the CORDIC constant and to perform the additions required in the diagonal cells. In total we need $n^2/2 + n/2$ AUs if we use one CORDIC unit per off-diagonal element or $3n^2/4$ if we use two.

The computation time for the CORDIC version of the BLV array will be computed under the same assumptions used for the floating point design. Namely, we will assume that U and V are computed with the unused portions of each processors cycle. We will also assume a data flow design. With these assumptions the computation time is given by the number of sweeps times (n-1) iterations per sweep times the number of OPs per iteration. As in the floating

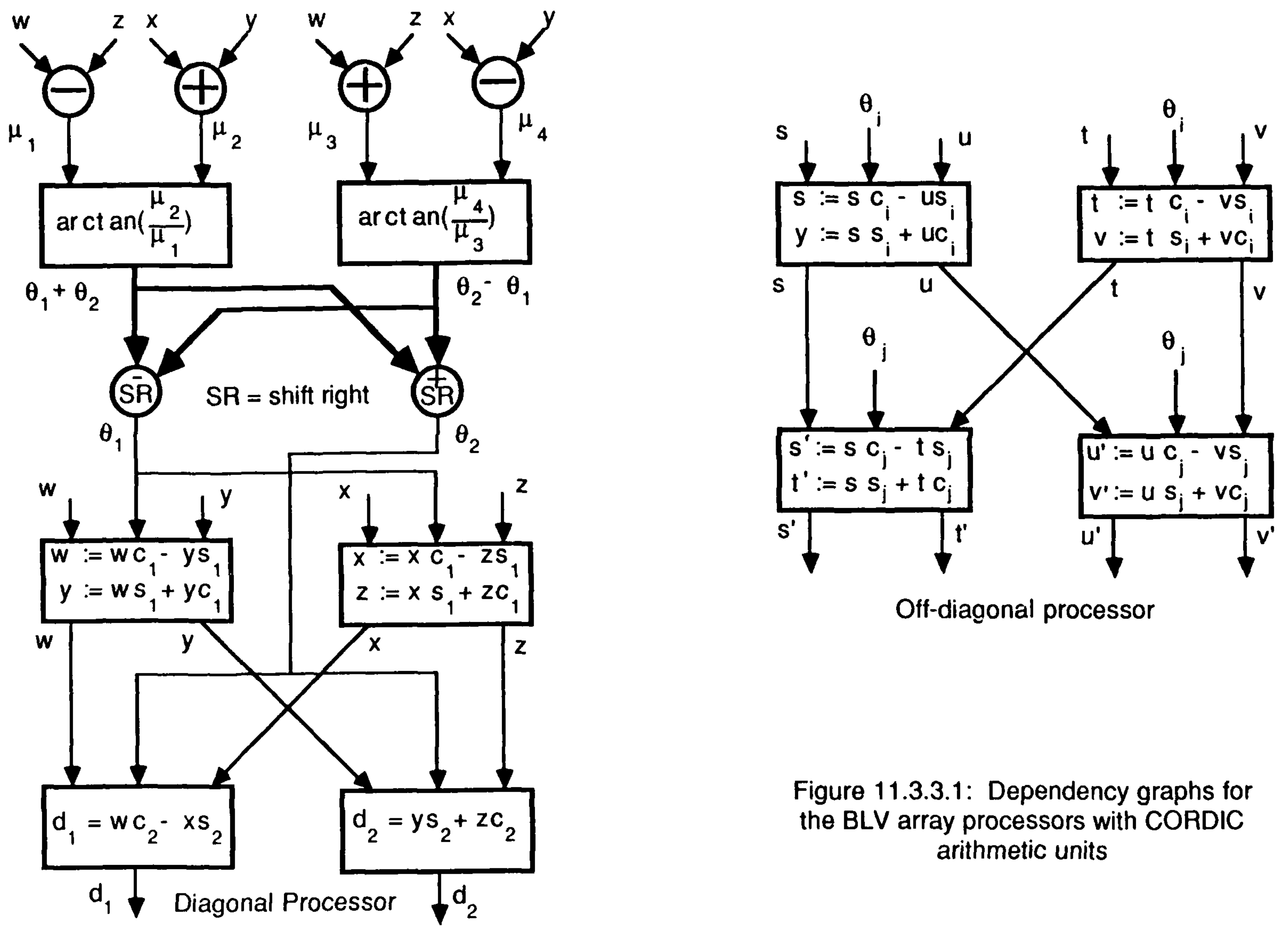


Figure 11.3.3.1: Dependency graphs for the BLV array processors with CORDIC arithmetic units

point case, each iteration consists of a three step cycle. During the first step the diagonal processors compute rotations and apply them to their 2-by-2 submatrices. During the other two steps the off-diagonal processors apply rotations. With two CORDIC AUs and one floating point AU allocated to a diagonal processor, the computation time for step one is 40 OPs. With one CORDIC AU and one floating point AU allocated to an off-diagonal processor, the computation time for steps two and three is 44 OPs each. This gives a total cycle time of 128 OPs which is somewhat longer than the 89 OPs of the floating point design. If we use two CORDIC AUs in each off-diagonal cell, the time for steps two and three is reduced to 26 OPs each. This gives a total cycle time of 92 OPs. This is a reduction of approximately 28% in the computation time at the expense of doubling the number of AUs. This does not seem to be a reasonable trade-off especially considering that the BLV array is already very expensive in terms of chip area. Therefore we will allocate only one CORDIC AU to each off-diagonal cell. Accordingly the SVD time for the CORDIC version of the BLV array ($T_{BLV(co)}$) is given by

$$T_{BLV(co)} = 128\,[3.1 \log_{10}(n) + 1.5]\,(n-1) \text{ OPs} \tag{11.3.3.1}$$

11.3.4 Luk Array

Exactly the same CORDIC modifications used in the BLV array can be applied to the Luk array. That is, we can use two CORDIC processors and one floating point AU in the diagonal elements and one CORDIC and one floating point unit in the off-diagonal processors. Also, we can still use the strategy used in the floating point version of pairing cells and using a set of AUs per pair. The CORDIC units in the diagonal processors would have to be modified to compute outer rotations. This is a simple modification involving sign and variable changes in the CORDIC equations. It does not change the CORDIC algorithm or the area requirement of a CORDIC unit. With this allocation, the total number of AUs in the CORDIC version of Luk's design is $n^2/2 + 3(n - 1)/2$, double the number of the floating point version.

The computation time is equal to the number of sweeps (including an extra 1/2 sweep for the QRD which can also be performed with CORDIC hardware) times n iterations per sweep times the number of OPs per iteration. Each iteration in the Luk array consists of a two step cycle, one step for the diagonal processors to compute and apply rotations and the other for the off-diagonal cells to apply them. With two CORDIC AUs and one floating point AU allocated to a diagonal processor, step one requires 38 OPs (two fewer than the BLV case since the sub-diagonal element of the 2-by-2 matrix is already zero in the Luk array). With one CORDIC AU and one multiplier per off-diagonal cell, step two requires 44 OPs. The total time for the two step cycle is 82 OPs. The overall computation time for the CORDIC version of Luk's array ($T_{Luk(co)}$) is given by

$$T_{Luk(co)} = 82\,[3.1 \log_{10}(n) + 2.0]\; n \text{ OPs} \tag{11.3.4.1}$$

This is slower than the floating point version.

11.3.5 Finn Array

The Finn architecture can not use CORDIC units efficiently. This is because we have concentrated on Finn's method C algorithm which uses cosine parameters which can not be easily computed by CORDIC units. We could use CORDIC processors in some of the other approximate Hestenes algorithms discussed by Finn [Fin83]. However these other algorithms have even slower convergence rates than Method C. Since the Finn architecture with Method C has already been shown to be non-competitive with the other architectures, it would be pointless to analyze an even slower algorithm.

11.4 Comparison of CORDIC a.d Floating Point Designs

In this section we will compare the resource requirements of the three architectures which have both CORDIC and floating point versions (Moreno, Luk and BLV). The intent of the comparison is to determine if the use of CORDIC processors changes the conclusions given in section 10.9 for the floating point designs. The comparisons are given again in the form of charts. In this case we give just one chart covering the values of n from 10 to 100. These charts are adequate to present both the asymptotic behavior for large matrices and the detailed characteristics for small matrices.

11.4.1 Number of AUs

Figure 11.4.1.1 shows the number of AUs required by the CORDIC versions in comparison to the floating point versions. We see that while the CORDIC version of Moreno's architecture requires fewer AUs, the BLV and Luk designs require twice as many. This is a direct result of the need for a multiplier to apply the CORDIC constant in each of the off-diagonal cells of the BLV and Luk arrays. As in the floating point case the linear array requires many fewer AUs than the quadratic arrays.

11.4.2 Computation Time

Figure 11.4.2.1 shows the number of OPs required by the CORDIC versions in comparison to the floating point versions. We see that the only significant change is the number of OPs in the Moreno architecture. Asymptotically, the

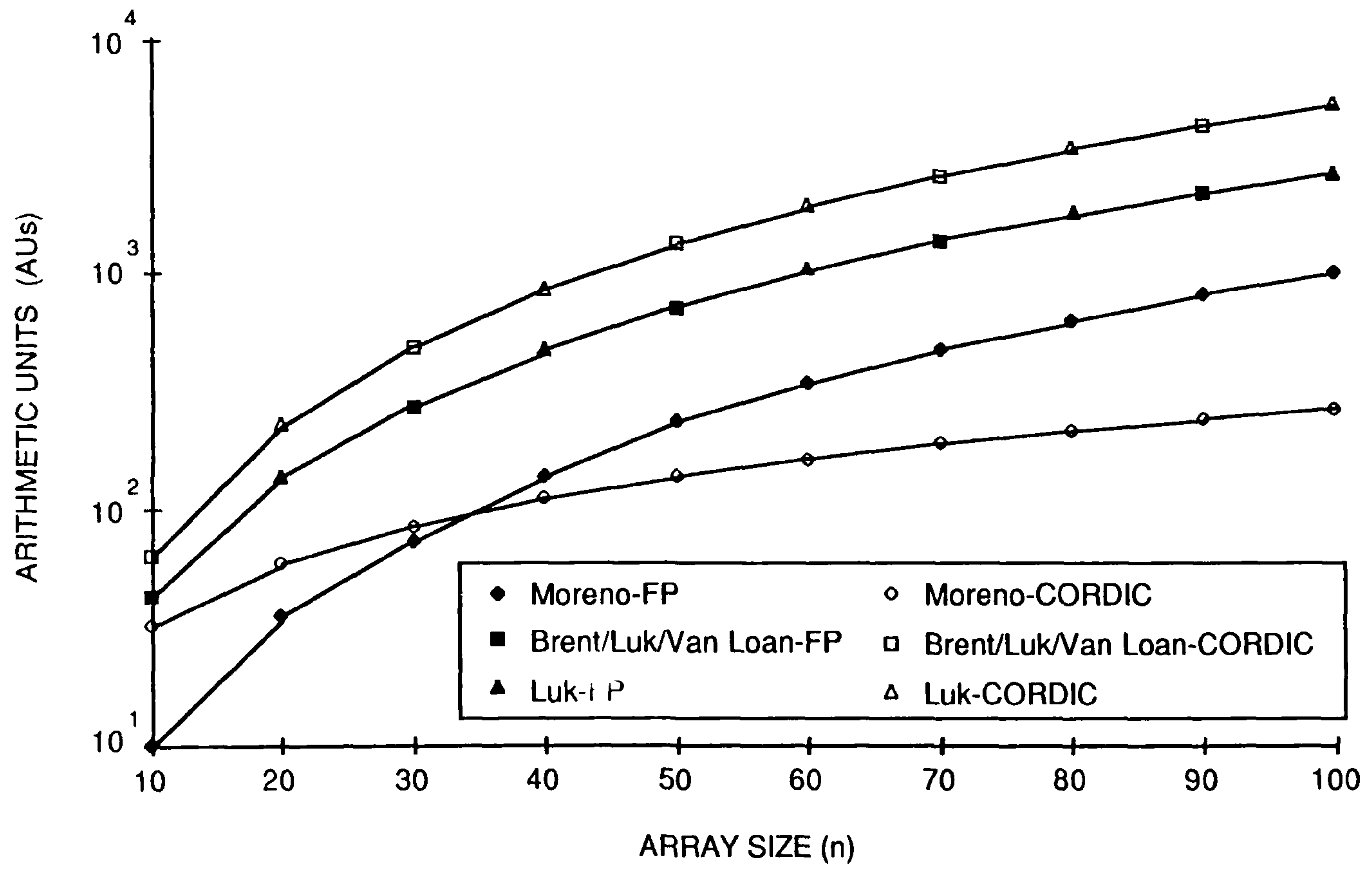


Figure 11.4.1.1: Arithmetic units required by CORDIC and floating point SVD architectures to compute U, Σ and V.

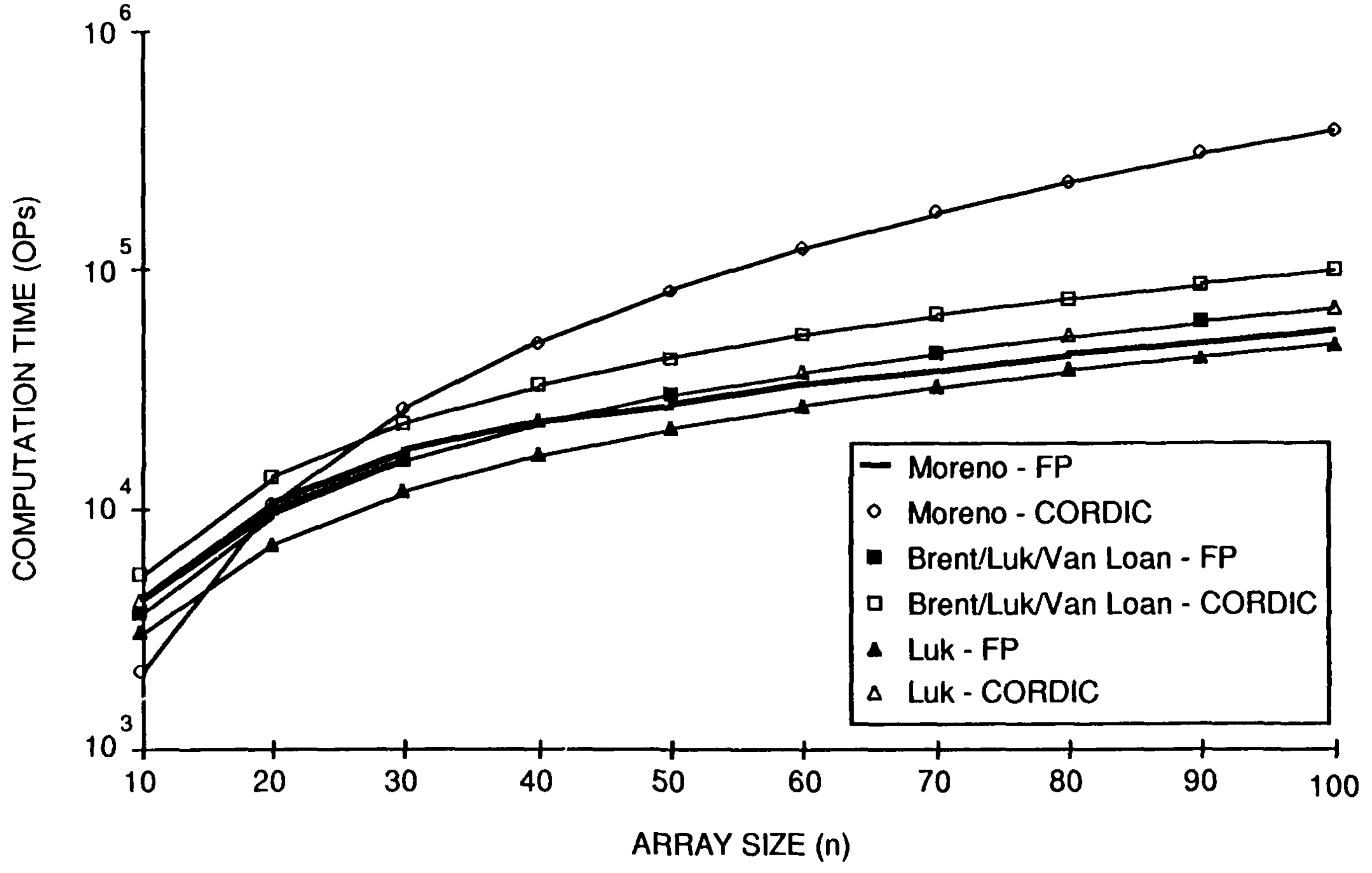


Figure 11.4.2.1: Computation time for U, Σ and V for CORDIC and floating point SVD architectures

CORDIC version requires 10 times more OPs. The CORDIC versions of both the BLV and Luk designs require a small constant more OPs than the floating point versions. The important observation for this chart is that the computation time for the quadratic arrays is now distinctly better than that of the linear array for all values of n > 30. In the floating point case the linear array was competitive for all values of n up to 200. The reason for this change is that the CORDIC units slow the Moreno array down by a factor of 10 but have only a small effect on the time for the quadratic arrays.

11.4.3 Total Resource Requirements

Figure 11.4.3.1 shows the total resource requirements (OPs x AUs) for the CORDIC designs in relation to the floating point versions. The chart shows that the CORDIC versions of all of the architectures are more expensive. The CORDIC versions of the BLV and Luk arrays have twice as many AUs and require more time per iteration. For the CORDIC version of the Moreno array the number of AUs decreases by a factor of approximately 6 but the computation time increases by a factor of 10. We see that the CORDIC version of Moreno's linear array is considerably less expensive than the CORDIC versions of both quadratic arrays as was the case for the floating point versions. Finally the chart shows that the CORDIC designs are still expensive in comparison to the Golub-Reinsch algorithm.

11.4.4 Efficiency

Figure 11.4.4.1 compares the efficiency of the CORDIC and floating point architectures. The chart shows that both versions of the Moreno architecture can be designed to give 100% efficiency. The efficiencies of the CORDIC versions of

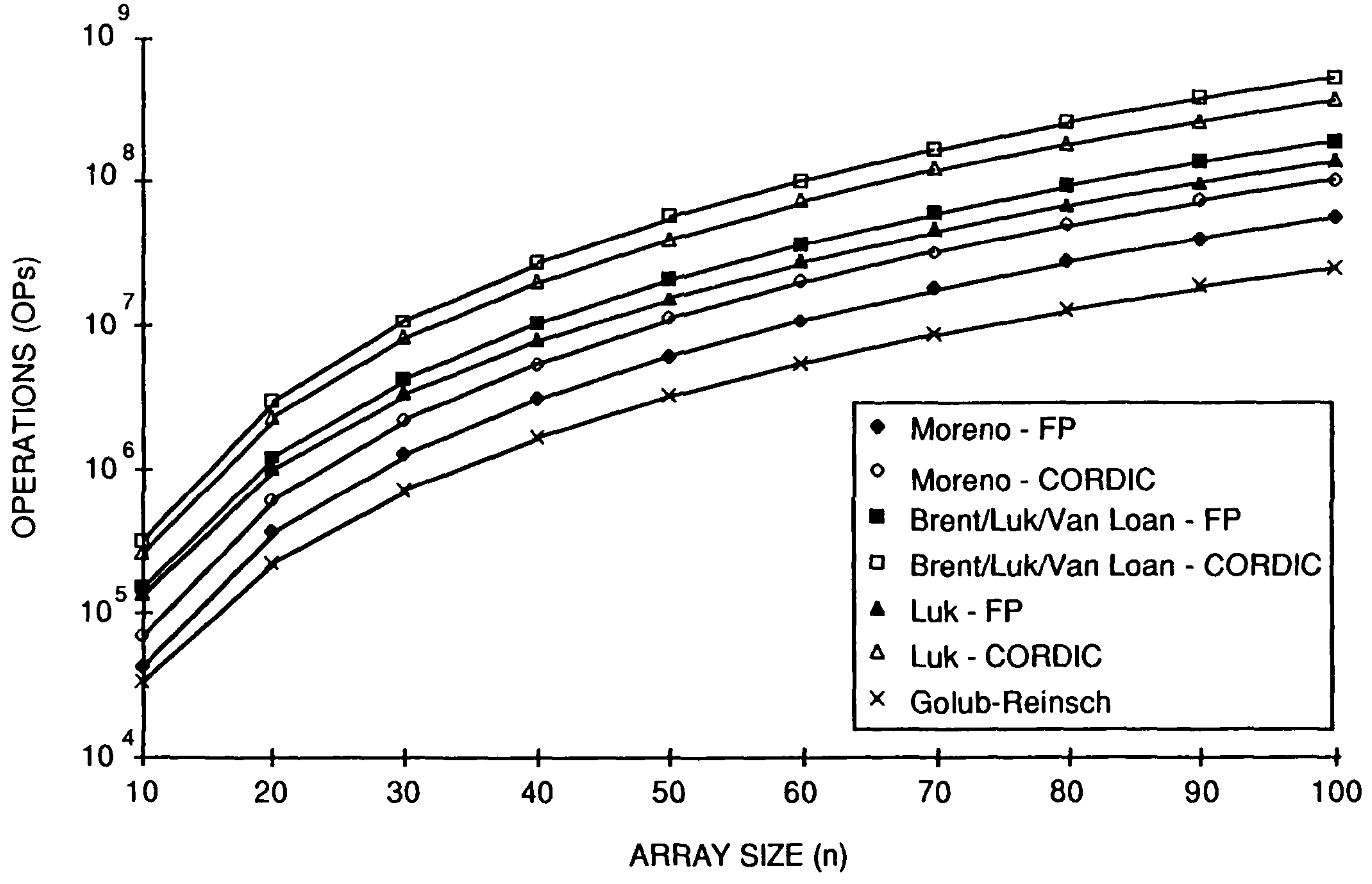


Figure 11.4.3.1: Total resource requirements (OPs x AUs) of CORDIC and floating point SVD architectures for the computation of U, Σ and V

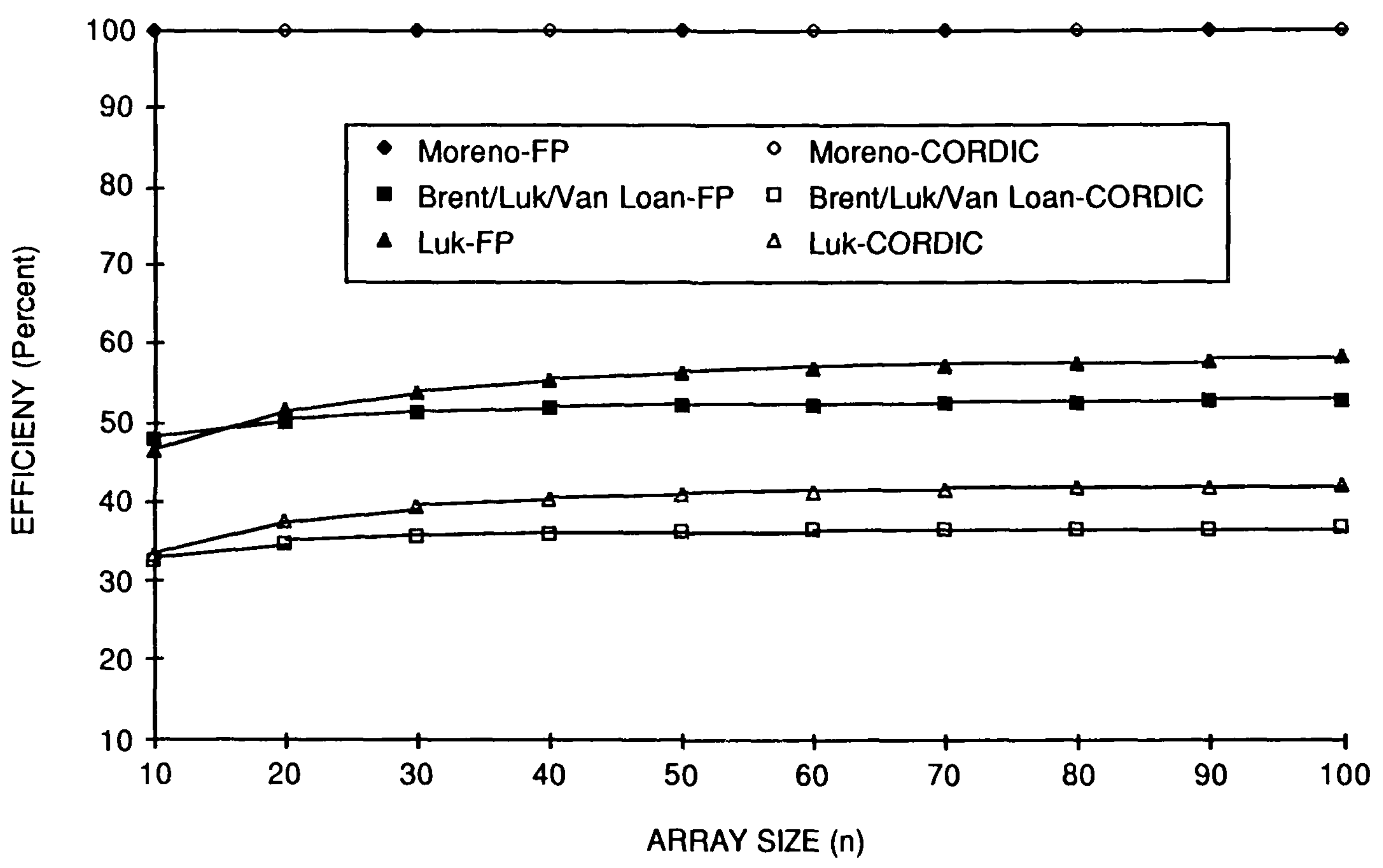


Figure 11.4.4.1: Efficiency of CORDIC and floating point SVD architectures for the computation of U, Σ and V.

both quadratic arrays is worse than the corresponding floating point efficiencies. This is because of the multipliers in the off-diagonal processors. Most of the time these AUs are sitting idle. For example in the CORDIC version of the BLV array the multipliers in the off-diagonal processors are active for only 8 OPS out of every 128 OP cycle or only 6% of the time. In contrast the CORDIC AUs in the off-diagonal cells are used very effectively (80 OPs per 128 OP cycle - 63%). Overall the CORDIC, BLV array has an asymptotic efficiency of 34%. For similar reasons the asymptotic efficiency of the CORDIC version of the Luk array is only 44%. The floating point versions of the BLV and Luk arrays were shown in section 10.5.1 to be 54% and 62% efficient, respectively.

11.4.5. Speedup

Finally, Figure 11.4.5.1 shows the speedup provided by the CORDIC designs in comparison to the speedup for the floating point designs. The chart shows that all of the CORDIC designs provided significantly lower speedup than the floating point designs. This is expected since the computation times of the CORDIC designs are all greater than the floating point designs.

11.5 Observations and Conclusions

The greatest benefit of the CORDIC units is the dramatic simplification of the cells in each of the architectures. This is especially true of the θ/NU unit in the Moreno design and the diagonal cells in the quadratic arrays. We can replace many floating point AUs, the tables for initial values for divisions and square-roots and much of the control structure of these units with a few simple CORDIC processors. Therefore the area requirement of the CORDIC designs should be lower than the floating point equivalents. Also because we only

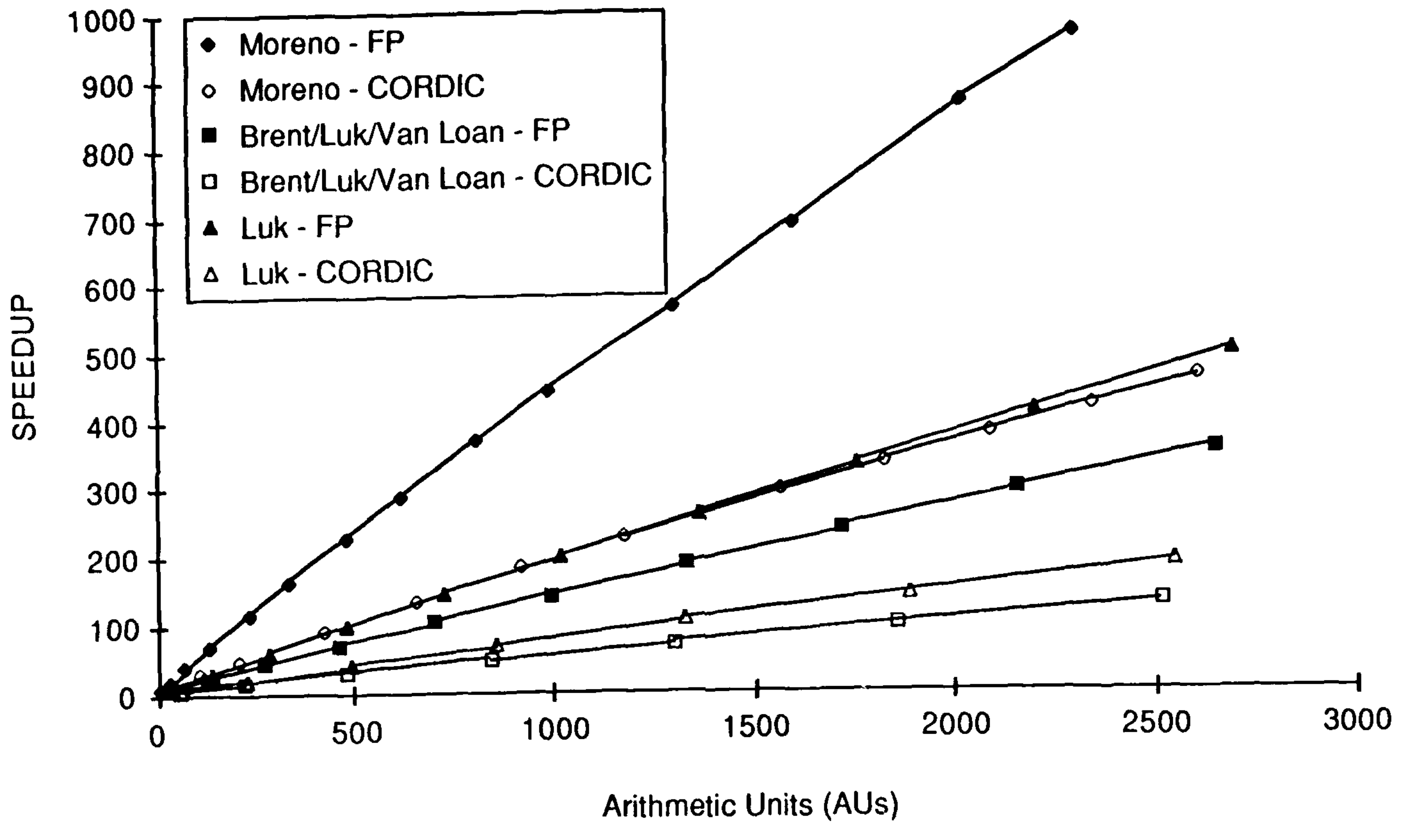


Figure 11.4.5.1: Speedup provided by CORDIC and floating point SVD architectures for the computation of U, Σ and V.

transmit one parameter (θ) the interconnect structure of the CORDIC designs is simpiified.

However, the comparison charts given in section 11.4 indicate that there is a price to pay for the simplification. The total resource requirements of all of the architectures is significantly higher with the CORDIC AUs. The primary difficulty in the Moreno design is the increase in computation time caused by the CORDIC AUs. In the quadratic arrays the problem is the need for the multipliers in the off-diagonal cells to apply the CORDIC constant. Without these multipliers the CORDIC versions of the quadratic arrays would be very competitive with the floating point versions. The number of AUs would be the same for either version and the CORDIC computation time would be only slightly longer. But because a CORDIC AU can not apply its own constant we are forced to include the extra hardware.

The charts show that the use of CORDIC processors makes the computation time of the quadratic arrays more competitive with the linear arrays for small matrices. However they do not change the relationship between the linear and quadratic arrays in terms of total resource requirements. The quadratic arrays still require more resources Finally the CORDIC processors do not increase the speedup provided by the architectures.

12.0 CONCLUSIONS

We have computed the number of bits needed in the arithmetic units of SVD arrays and we have analyzed and compared the resource requirements of several proposed SVD architectures.

Our results are based on the assumption that we are operating on matrices of quantized data. This assumption is realistic for many digital signal processing applications of the SVD. Matrices of 8-bit and 16-bit quantized data values are quite common in image processing and seismic and hydroacoustic data processing. Many other applications are also covered since it is common practice to generate digital data values with finite precision A-to-D converters.

The assumption that we are operating on quantized data elements is crucial to our results. Since the data elements have quantization errors, the singular values and singular vectors generated by the SVD will have "quantization errors" also. Therefore it is unnecessary and, in fact, unrealistic to compute the SVD to extremely high precision. In computing the number of bits needed in the AUs of an SVD array, we have assumed that we must have enough bits to keep the magnitude of the largest round-off error at or below the magnitude of the quantization error.

We have shown both theoretically and experimentally that the variance of the quantization error of the singular values of a quantized data matrix is as large as the variance of the quantization error of the data. This result is not only important for computing the number of bits needed in the AUs, but it is also significant by itself. It shows that we must be aware of the the characteristics of the original data in deciding how to use the results of an SVD computation. We must be very wary of using singular values which are smaller than, say, two times the standard deviation of the quantization error.

We used the analysis given by Wilkinson [Wil65] for the symmetric eigenvalue problem to bound the magnitude of the round-off error for the Jacobi and Hestenes SVD algorithms. Wilkinson's analysis covers standard fixed point and floating point arithmetic. Following Wilkinson's arguments, we developed a similar bound for the round-off error of the algorithms using fixed point, CORDIC AUs. The bounds for CORDIC arithmetic are very similar to Wilkinson's bound for fixed point arithmetic but include an additional factor of "t" to account for the number of iterations needed to complete a CORDIC computation.

Our simulations of the SVD algorithms with finite precision arithmetic showed the theoretical bounds to be much too loose. Based on a careful analysis of the accumulation of errors in the simulation runs, we were able to develop much tighter approximate, statistical bounds. For standard and CORDIC fixed point arithmetic the new bounds are $O(n^{3/2})$ lower than the theoretical bounds (for square matrices). For the floating point case, the reduction is $O(\sqrt{n})$.

We used the statistical bounds to compute the number of bits needed in SVD array AUs to insure that the round-off error is no larger than the quantization error. Our results show that we need essentially the same number of bits for either the Hestenes or Jacobi algorithms (when processing square matrices). If we use properly rounded shift and add operations, CORDIC processors require approximately 8 fewer bits than floating point AUs. Our computations indicate that once the input matrix has been normalized to prevent overflows, standard fixed point AUs can be used very effectively in the rotation application units of SVD array. In fact, fewer bits are required by the fixed point AUs than either the CORDIC or floating point AUs for the application of rotations. Finally our computations show that fairly large words are needed to compute the SVD accurately if we are processing large matrices or input data with many bits. For example, 32 bit floating point AUs are useful only for small arrays of 8-bit data. For applications involving the decomposition of 100-by-100 arrays of 16-bit data we will need 40-bit floating point AUs. However, we do see that commonly available 32-bit fixed point AUs could be used in the off diagonal processors of SVD arrays designed for large 8-bit matrices or moderate size 16-bit arrays.

We have described five different SVD architectures and compared their resource requirements with floating point and CORDIC arithmetic units. The comparison shows that the total resource requirements (OPs x AUs) of the linear designs are lower than that of the quadratic arrays for all size matrices. With floating point AUs, the difference between the best quadratic array (Luk's) and the linear designs is asymptotically a factor of 2.5. In addition, the quadratic arrays require much more area per AU due to the overhead of communications, control, and data storage.

We have seen that the computation time of the linear arrays is competitive with that of the quadratic arrays for matrices up to size 200-by-200. The linear arrays perform so well because they use the Hestenes algorithm and apply their operations very efficiently. The BLV and Luk arrays use the Jacobi algorithm, which requires more operations, and both designs are less than 62% efficient. The Finn design is not competitive with the others due to the increased number of sweeps required by its approximate Hestenes algorithm.

We have also seen that the architectures can provide a significant speedup in the computation of the SVD. This is really what we are after in developing such complex structures. The best speedup is obtained when we want to compute U, Σ and V of a dense matrix. If we need only Σ, the speedup provided by these architectures is poor in comparison to the Golub-Reinsch algorithm.

Finally we have seen that CORDIC arithmetic units can greatly simplify the portions of the architectures which compute rotation parameters since they eliminate the complex division and square root operations. However the CORDIC processors effectively double the total resource requirements of all of the architectures. They also result in longer computation times. With the recent announcement of a floating point chip which includes divisions and square roots as single clock-cycle instructions, it appears that CORDIC based SVD arrays will not be the wave of the future.

It is clear that parallel SVD architectures will be very useful for signal processing applications. They offer computation times which will allow the SVD to be used in other than a batch processing mode. Some of the designs could be constructed today for reasonable size matrices ($n \approx 100$). In particular the Schimmel/Luk design could be fabricated easily with presently available 32/64-bit floating point ALU's and 32-bit fixed point multiply-accumulate chips. However, full application of these architectures, particularly the quadratic arrays, to large matrices ($n \geq 1000$) will have to wait for improvements in VLSI technology and the introduction of wafer scale integration.

References

[Ahm81] Ahmed, H. M., Signal Processing Algorithms and Architectures, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Dec. 1981.

[And67] Anderson, S. F., Earlie, J. G., Goldschmidt, R. E and Powers, D. M., "The IBM System/360 Model 91: Floating Point Execution Unit," IBM Journal, Jan. 1967, pp. 34-53.

[And76] Andrews, H. and Patterson C., "Singular Value Decomposition and Digital Image Processing," IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-24(1), Feb. 1976, pp. 26-53.

[Bre84] Brent, R. P. and Luk, F. T., "The Solution of Singular Value Problems Using Systolic Arrays," SPIE Real Time Signal Processing VII, Vol. 495, 1984, pp. 7-12.

[Bre85a] Brent, R. P., Luk, F. T. and Van Loan, C., "Computation of the Singular Value Decomposition Using Mesh Connected Processors," Journal of VLSI and Computer Systems, Vol. 1, 1985, pp. 242-270.

[Bre85b] Brent, R. P. and Luk, F. T., "The Solution of Singular-value and Symmetric Eigenvalue Problems on Multiprocessor Arrays," SIAM Journal of Scientific and Statistical Computing, Vol. 6, 1985, pp. 69-84.

[Cav86] Cavallaro, J.R. and Luk, F.T., "Architectures for a CORDIC SVD Processor," SPIE Real Time Signal Processing IX, Vol. 698, 1986 pp. 45-53.

[Cav87] Cavallaro, J.R. and Luk, F.T., "CORDIC Arithmetic for an SVD Processor," Proceedings of the 8th Symposium on Computer Arithmetic, 1987, pp. 113-120.

[EDN87] "Product Update - Floating-point Chip Set Executes 60M Flops," EDN, March 4, 1987, pp. 95-96.

[Fin82a] Finn, A. M., Luk, F. T. and Pottle, C.. Systolic Array Computation of the Singular Value Decomposition," SPIE Real Time Signal Processing V, Vol. 341, 1982, pp. 35-43.

References (continued)

[Fin82b] Finn, A. M. and Pottle, C., "An Algorithm and Simulation Results for a Systolic Array Computation of the Singular Value Decomposition," International Large Scale Systems Symposium, 1982, pp. 93-97.

[Fin83] Finn, A.M., Computation of the Singular Value Decomposition on a Quadratic Array of Processors, Ph.D. dissertation, School of Electrical Engineering, Cornell University, May 1983, University Microfilms Intl., Ann Arbor, MI.

[For60] Forsythe, G. E. and Henrici, P., "The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix," Transactions of the American Mathematical Society, Vol. 94, 1960, pp.1-23.

[Gol65] Golub, G. H. and Kahan, W., "Calculating the Singular Values and Pseudo-Inverse of a Matrix," Siam Journal on Numerical Analysis, Vol. 2, 1965, pp. 205-224.

[Gol70] Golub, G. H. and Reinsch, C., "Singular Value Decomposition and Least Squares Solutions," Numerische Mathematik, Vol. 14, 1970, pp. 403-420.

[Gol83] Golub, G. H. and Van Loan, C., Matrix Computations, The Johns Hopkins University Press, Baltimore, 1983.

[Hav80] Haviland, G. and Tuszynski, A., "A CORDIC Arithmetic Processor Chip," IEEE Journal of Solid State Circuits, Vol. SC-15, No.1, Feb. 1980, pp. 4-14.

[Hes58] Hestenes, M., "Inversion of Matrices by Biorthogonalization and Related Results," Journal SIAM, Vol. 6, 1958, pp. 51-90.

[Hu86] Hu, Y., "The Quantization Effects in a VLSI CORDIC Processor," Technical Report,Department of Electrical Engineering , Southern Methodist University, Dallas, Texas (in review to be published in IEEE Transactions on ASSP).

[Lei87] Leibson, S.H., "Microprogrammable Processing," EDN, April 15, 1987. pp. 143-160.

[Luk80] Luk, F., "Computing the Singular Value Decomposition on the Illiac IV," ACM Transactions on Mathematical Software, Vol. 6, 1980, pp. 524-539.

References (continued)

[Luk86] Luk, F., "A Triangular Processor Array for Computing Singular Values", Linear Algebra and Its Applications, Vol 77, 1986, pp. 259-273.

[Mor85] Moreno, J. H., Analysis of Alternatives for a Singular Value Decomposition Processor, M.S. Thesis, Dept. of Computer Science, University of California at Los Angeles, 1985.

[Opp75] Oppenheim, A., and Schafer R., Digital Signal Processing, Prentice-Hall, Englewood Cliff, NJ, 1975.

[Ram72] Ramamoorthy, C., Goodman, J. and Kim, K., "Some Properties of Iterative Square-Rooting Methods Using High Speed Multiplication," IEEE Transactions on Computers, Vol. C-21(2), Feb. 1972, pp. 137-146.

[Sch86] Schimmel, D. E. and Luk, F. T., "A New Systolic Array for the Singular Value Decomposition," Proceedings of the Fourth MIT Conference on Advanced Research in VLSI, pp. 205-217, April 1986.

[Sch86b] Schimmel, D. E., private conversation, April 1986.

[Shi81] Shim, Y. and Cho, Z., "SVD Pseudoinverse Image Reconstruction," IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-29(4), Aug. 1981, pp. 904-909.

[Spe83] Speiser, J. and Whitehouse, H., "A Review of Signal Processing with Systolic Arrays," Proceedings SPIE Vol 431, Real Time Signal Processing VI , 1983, pp. 2- 6.

[Ste85] Stewart, G. W., "A Jacobi-like Algorithm for Computing the Schur Decomposition of a non-Hermitian Matrix," SIAM Journal of Scientific and Statistical Computing, Vol. 6, 1985, pp.853-864.

[Van85] Van Loan, C., The Block Jacobi Method for Computing the Singular Value Decomposition, TR 85-680, Dept. of Computer Science, Cornell University, Ithaca, NY, June 1985.

[Vol59] Volder, J., "The CORDIC Trigonometric Computing Technique," IRE Transactions on Electronic Computers, Vol. EC-8, Sep 59, pp. 330-334.

References (continued)

[Vom83] vom Scheidt, J. and Purkert, W., Random Eigenvalue Problems, Elsevier Science Publishing Co., New York, NY, 1983.

[Wal71] Walther, J., "A Unified Algorithm for Elementary Functions," Spring Joint Computer Conference, 1971, pp. 379-385.

[Was82] Waser, S. and Flynn, M.J., Introduction to Arithmetic for Digital Systems Designers, Holt, Rinehart and Winston, New York, NY, 1982.

[Wil65] Wilkinson, J.H., The Algebraic Eigenvalue Problem, Clarendon Press, Oxford, 1965.

[Yoh73] Yohe J. M., "Roundings in Floating Point Arithmetic," IEEE Transactions on Computers Vol C-22, No. 6, June 1973, pp. 577-586.